Aplicación de Big Data al análisis, monitorización y seguridad de redes de comunicaciones

Por

RAFAEL LEIRA OSUNA

UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Departamento de Tecnología Electrónica y de las Comunicaciones

TESIS DOCTORAL

OCTUBRE DE 2019

Director: Dr. Iván González

Francisco Tomás y Valiente, 1
Madrid, 28049
Spain

RAFAEL LEIRA OSUNA
Aplicación de Big Data al análisis, monitorización y seguridad de redes de comunicaciones

Escuela Politécnica Superior.
High Performance Computing and Networking Research Group

IMPRESO EN ESPAÑA PRINTED IN SPAIN

Departamento: Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Tesis doctoral: Aplicación de Big Data al análisis, monitorización y seguridad de redes de comunicaciones

Autor: Rafael Leira Osuna

Director: Dr. Iván González

Año: 2019

Tribunal:

Presidente Dr. José Alberto Hernández Gutiérrez

Secretario Dr. José Luis García Dorado

Vocal Dr. Víctor López Álvarez

Sustituto Dr. Francisco Javier Ramos de Santiago

Sustituto Dr. Víctor Moreno Martínez

Esta tesis ha sido realizada dentro del grupo *High Performance Computing and Networking research group*, en el Departamento de Tecnología Electrónica y de las Comunicaciones, Escuela Politécnica Superior, Universidad Autónoma de Madrid. Durante los primeros cuatro años de la realización de la tesis se recibieron los siguientes contratos cuyo objetivo principal era la ayuda a la realización de la tesis: Investigador financiado mediante el proyecto Europeo EINS (un mes), FPI proporcionada la Universidad Autónoma de Madrid (2 años), FPU proporcionada por el Ministerio de Educación, Cultura y Deporte (MECD) español (2 años). El último año de la tesis se realizó como empleado de la empresa Naudit HPCN S.L. Esta tesis fue parcialmente financiada por el gobierno español (MINECO/FEDER: TEC2012-33754 y TEC2015-69417-C2-1-R); y por la Unión Europea a través del proyecto FP7-317999 y las concesiones 288021 (Proyecto Network of Excellence in Internet Science (EINS)) y 761727 (Proyecto METRO-HAUL) dentro del programa H2020.

Agradecimientos

Hay muchas personas sin las cuales esta tesis no podría haberse completado. El primero al que debo agradecer es a Iván González, mi magnífico y comprensivo tutor que me ha ayudado y compartido los mejores y peores momentos del desarrollo de esta tesis. Todo agradecimiento se queda pequeño. Un trocito de esta tesis es tuyo.

A Javier Aracil, que ha sido como un segundo tutor y gracias a sus reuniones con Carlos Vega y Paula Roquero surgieron grandes ideas y trabajos. Una de las muchas cosas buenas de Javier es su habilidad para abrir los ojos y mostrar nuevas perspectivas en los buenos y malos momentos. Gracias por abrirme los ojos tantas veces.

A Jorge Enrique López de Vergara Méndez, que gracias a sus conocimientos, disciplina y rigor logré obtener mi contrato FPU y mover un pasito más lejos el conocimiento humano.

A Diego Hernando que, aunque nos veamos poco, siempre nos hemos apoyado en lo posible. Y al resto del equipo de Handbe: Sin vosotros no podría haber aprendido grandes lecciones vitales. Lástima que las cosas acabasen así.

A Raúl Martín, una de esas personas que al conocerla es imposible que no te cambie un poquito como ves la vida.

A Sergio Fuentes (Cariñosamente llamado otouto), el cual ha hecho cada día un poquito más llevadero que el anterior.

A José Fernando Zazo, cuyas horas compartidas nos enseñaron mucho a ambos y desde luego hicieron este doctorado mucho más llevadero.

A Mario Ruiz, sin sus ideas y ayuda con las FPGAs esta tesis tendría un hueco insustituible.

A Guillermo Julián que además de haber colaborado a nivel técnico no podré olvidar las risas que nos echábamos codificando juntos.

A ese viajecito por Europa en el que me acompañaron Sergio y Guillermo, sin el cual puede que no hubiese conseguido las fuerzas para cerrar este capítulo de mi vida.

Y en general a todos los compañeros del grupo de investigación HPCN y de Naudit, ya que en mayor o menor medida han puesto su granito de arena a lo largo del Doctorado. Menciono especialmente a: David Muelas, Rubén García-Valcárcel, Isabel García, Tobías Alonso, Dixon Salcedo, Daniel Perdices y a Sergio López Buedo.

A mi familia, padres, tíos y primos que tan abandonados os he tenido los últimos años.

A Bea, que durante tantos años me ha aguantado y apoyado con fuerza en los buenos y malos momentos a pesar del estrés.

De la misma manera agradezco el apoyo a todas y cada una de las personas que he conocido a lo largo de la carrera, del Máster y del doctorado, a mis amigos de toda la vida que de una u otra forma han estado conmigo a lo largo de estos 4 largos años de doctorado haciéndolo más ameno y finalmente llevándome a escribir este tesis, y con ella, a finalizar mi doctorado.

Abstract

DURING the last two decades, the growth of networks has been exponential, both in complexity and speed. Internet has caused a social change that even almost 40 years of its appearance still has an undeniable impact on the evolution of technology and society. This growth and impact has generated the emergence of new business models focused on the Internet, as well as the need to move existing businesses to a digital model in the cloud. Today, the network has become a vital component both socially and economically. The investment of network infrastructure by companies is very high both at the Capital expenditures (CAPEX) and at the Operational Expenditures (OPEX). Protecting the investment of the network equipment together with the business model has made network monitoring a necessary process not only at the level of the Internet Service Provider (ISP), but at any size company level.

The objective of network monitoring is to obtain relevant information about the current state of the network and its components. This process is linked to the technology used in the network, both at the physical link level and at the equipment level. This relationship has forced monitoring technologies to be dependent on the monitored technology and, therefore, to have to redesign the systems with each new technological leap. Partially or completely redesigning the hardware and software of the monitoring system has caused these systems not only to be commercially expensive, but to be delayed for 2 to 5 years with respect to the latest available network technology. This effect is not due to a problem of standardization of equipment between technological leaps, but to the challenge of managing the increase in information per second that is transmitted and generated in the monitored network devices.

This thesis arises from the need to build a comprehensive monitoring scheme and model using the latest Big Data techniques, with the aim of offering a system capable of evolving with network technology, lowering the costs of futu-

re developments and reducing space between a new transmission technology and the development of its associated monitoring system. At the same time, the second objective is to unify the monitoring process, from the sampling of network information, through the processing and analysis of the data to the final and summarized visualization of the relevant parameters of the network status. In this way, we define an integral monitoring as an application divided into 4 fundamental processes: sampling, preprocessing and filtering, analysis and visualization.

There are fundamentally two ways to obtain information from the network: passive monitoring and active monitoring. In this thesis new ideas are explored to perform both active and passive measurements and with different methods focused on software and virtualization. Easily portable scalable sampling solutions are proposed to future technologies, and with the ability to obtain equivalent results and in some cases, superior to the state of the art in their respective fields.

The preprocessing and filtering stage becomes a complex task when handling data from passive monitoring, the most expensive in terms of computing and storage requirements, since it is potentially necessary to handle more than 100 million messages per second in a 100 Gbit/s network link. After carefully evaluating the different Big Data solutions for handling distributed communications, pre-filtering and processing, we observe that no previous system is capable of obtaining the necessary performance, unless tens or even hundreds of equipment dedicated exclusively to this task are used. For this reason, a proprietary tool for high-speed distributed communications has been developed, which has been called Wormhole, which offers a transmission and computation capacity well above current systems. The generality of Wormhole allows you to integrate different low-level traffic analysis tools to simplify the flow-level information in real time, and store the information to later perform more complex analyzes through offline processes.

Thanks to the previous two stages, the information is already distributed, which allows the analysis stage to operate on a smaller data set, and perform a complex analysis in a simple way. In this thesis, we present several examples of complex network data analysis such as deep packet inspection (DPI), anomaly detection through unsupervised learning models, and classification of network flows using deep neural networks. These are case studies that demonstrate the potential of the proposed solution. The visualization stage can be easily addressed by using existing open source solutions such as Grafana, Kibana, etc. This stage allows to show the results of the monitoring and analysis, as well as interacting

with the end user of the monitoring system. The development of custom dashboards would be the right solution, however, it is not the objective of this thesis.

Finally, together with all the stages and solutions proposed in each of them, the thesis presents a unique and comprehensive monitoring system whose design, based on different Big Data solutions, allows monitoring more complex networks by simply increasing the number of nodes in the system or the computing power of (a subset of) the nodes.

Resumen

DURANTE las dos últimas décadas, el crecimiento de las redes ha sido exponencial, tanto en complejidad como en velocidad. Internet, como la mayor red de computadores a nivel global, ha causado un cambio social que incluso casi 40 años desde su aparición sigue teniendo un impacto innegable en la evolución de la tecnología y la sociedad, y ha generado la aparición de nuevos modelos de negocio, así como la necesidad de trasladar negocios existentes a un modelo digital en la nube. A día de hoy, la red se ha vuelto un componente vital tanto a nivel social como a nivel económico. La inversión de infraestructura de red por parte de las empresas es muy elevada tanto a nivel de equipamiento inicial (CAPEX) como a nivel de mantenimiento y operación (OPEX). Proteger esta inversión, junto con el modelo de negocio, ha convertido a la monitorización de red en un proceso necesario ya no solo a nivel de operadora de servicios de internet (ISP), sino a nivel de empresa de cualquier tamaño.

El objetivo de la monitorización de la red es la obtención de información relevante acerca del estado actual en el que se encuentra la red, y de los componentes que la forman. Este proceso está ligado a la tecnología utilizada en la red, tanto a nivel de enlace físico como a nivel de componente. Esta relación ha obligado a las tecnologías de monitorización a ser dependientes de la tecnología monitorizada y, por tanto, a tener que rediseñar los sistemas con cada nuevo salto tecnológico. Rediseñar parcial o completamente el hardware y software de los sistemas de monitorización ha causado que estos sistemas no solo sean comercialmente caros, sino que se encuentren retrasados de 2 a 5 años con respecto a la última tecnología de red disponible. Este efecto no es debido a un problema de estandarización del equipamiento entre saltos tecnológicos, sino al reto que supone manejar el incremento de información por segundo que se transmite y genera en los dispositivos de red monitorizada.

Esta tesis surge de la necesidad de construir un esquema y modelo de monitorización integral utilizando las últimas técnicas de Big Data, con el objetivo de ofrecer un sistema capaz de evolucionar con la tecnología de la red, abaratando los costes de futuros desarrollos y reducir el espacio entre una nueva tecnología de transmisión y el desarrollo de su sistema de monitorización asociado. A su vez se plantea como segundo objetivo unificar el proceso de monitorización, desde el muestreo de información de la red, pasando por el procesamiento y análisis de los datos hasta la visualización final y resumida sobre los parámetros relevantes del estado de la red. De esta forma, definimos una monitorización integral como una aplicación dividida en 4 procesos fundamentales: muestreo, preproceso y filtrado, análisis y visualización.

El proceso de muestreo es el componente más complejo técnicamente hablando y más dependiente de la tecnología subyacente. Por ello, en muchas ocasiones se le ha denominado directamente "monitorización". Existen fundamentalmente dos formas de obtener información de la red: monitorización pasiva y monitorización activa. En esta tesis se exploran nuevas ideas para realizar mediciones tanto activas como pasivas y con diferentes métodos centrados en software y virtualización. Se proponen soluciones escalables de muestreo fácilmente portables a futuras tecnologías, y con la capacidad de obtener resultados equivalentes y en algunos casos, superiores al estado del arte en sus respectivos campos.

La etapa de preprocesado y filtrado se vuelve una tarea compleja a la hora de manejar los datos provenientes de la monitorización pasiva, la más costosa a nivel de requerimientos de cómputo y almacenamiento, pues potencialmente es necesario manejar más de 100 millones de mensajes por segundo en un enlace de red de 100 Gbit/s. Tras evaluar cuidadosamente las diferentes soluciones Big Data para el manejo de comunicaciones distribuidas, pre-filtrado y procesado, observamos que ningún sistema previo es capaz de obtener el rendimiento necesario, salvo que se empleen decenas o incluso cientos de equipos dedicados en exclusiva a este cometido. Por este motivo se ha desarrollado una herramienta propia orientada a comunicaciones distribuidas de alta velocidad, a la que se ha llamado Wormhole, que ofrece una capacidad de transmisión y cómputo muy por encima de los sistemas actuales. La generalidad de Wormhole le permite integrar diferentes herramientas de análisis de tráfico de bajo nivel para simplificar la información a nivel de flujo en tiempo real, y almacenar la información para posteriormente realizar análisis más complejos mediante procesos offline.

Gracias a las dos etapas previas la información ya se encuentra distribuida, lo que permite a la etapa de análisis operar sobre un conjunto de datos más reducido, y realizar un análisis complejo de forma sencilla. En esta tesis presentamos varios ejemplos de análisis de datos de red complejos como son la inspección profunda de paquetes (DPI), la detección de anomalías mediante modelos de aprendizaje no supervisado, y la clasificación de flujos de red mediante redes neuronales profundas. Se trata de casos prácticos que demuestran el potencial de la solución propuesta.

La etapa de visualización se puede abordar fácilmente mediante el uso de soluciones ya existentes de código abierto como Grafana, Kibana, etc. Esta etapa permite mostrar los resultados de la monitorización y análisis, así como interaccionar con el usuario final del sistema de monitorización. El desarrollo de dashboards a medida sería la solución adecuada, sin embargo, no es el objetivo de esta tesis.

Finalmente, unidas todas las etapas y soluciones propuestas en cada una de ellas, la tesis presenta un único e integral sistema de monitorización cuyo diseño, basado en diferentes soluciones Big Data permite monitorizar redes más complejas con solo incrementar el número de nodos del sistema o en su defecto, la potencia de cálculo de los mismos.

Tabla de Contenidos

Página

Índice de tablas

TABLA **Página**

Índice de figuras

FIGURA **Página**

1 Introducción

La tecnología ha transformado como entendemos el mundo que nos rodea. Los seres humanos hemos buscado durante toda nuestra historia mejorar nuestros métodos y canales de comunicación. Esta necesidad ha propiciado la aparición y evolución de diferentes dispositivos a lo largo de la historia como la imprenta, el telégrafo, el teléfono, los ordenadores o los móviles. Cada uno de ellos, ha ido acompañado de nuevos paradigmas de comunicación y nos han permitido acelerar nuestra evolución como sociedad, acortando no solo las distancias que separan a los seres humanos entre sí, sino que poco a poco han globalizado y facilitado la información.

Internet es el producto de esta necesidad y probablemente uno de los mayores logros tecnológicos del siglo pasado. Hace más de tres décadas que se publicó el estándar de Internet Protocol (IP) [1]. Sin embargo, apenas 37 años más tarde, se estima un tráfico mensual de 150 Exabytes de tráfico IP en todo el mundo, con unas expectativas de crecimiento de un 85 % para 2021 [2]. Este incremento en el volumen de datos a nivel IP se debe principalmente a dos factores: (I) el incremento de dispositivos que se conectan a Internet (como móviles, relojes, neveras [3], etc.) y (II) el avance tecnológico de los medios de transmisión.

Este hecho se puede observar de forma gráfica gracias a la ley de Nielsen, que pronostica un incremento en la velocidad de acceso a Internet a un ritmo de crecimiento de un 50 % anual. Una ley que aparentemente se ha cumplido de acuerdo a los datos proporcionados por Nielsen desde 1983 (ver Figura 1.1).

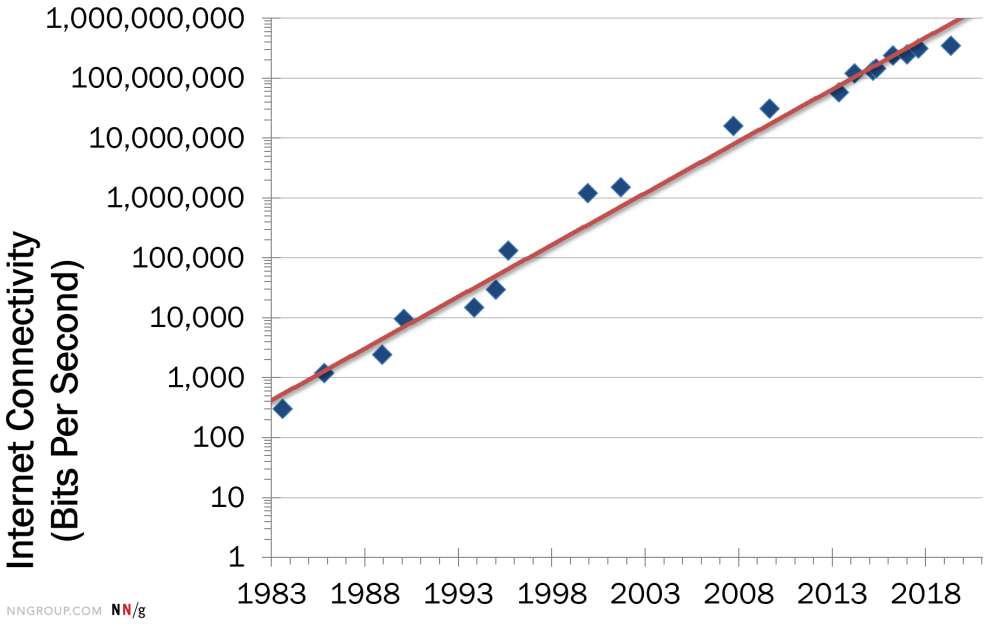


Figura 1.1. Evolución de la velocidad de acceso a Internet

Actualmente en España, es posible contratar con facilidad enlaces de 100, 300 e incluso 600 Mbit/s en los hogares, lo que se traduce potencialmente en un ancho de banda agregado, a nivel de operadora, de varias decenas o incluso centenas de gigabits por segundo en tan solo un barrio. Hace 15 años, manipular información a una tasa de 1 Gbit/s se consideraba un reto de cómputo [4, 5], mientras que apenas 10 años más tarde, el reto tecnológico ha pasado de los 10 Gbit/s [6, 7], y se encuentra en los 40 Gbit/s y 100 Gbit/s. Y en breve, habrá que empezar a plantear el manejo de enlaces a 400 Gbit/s, ya existentes en grandes centros de procesamiento de datos y operadoras de Internet [8]. Es importante tener en cuenta que, al hablar de la velocidad en un CPD, nos solemos referir principalmente a la conectividad interna, en donde se suelen usar tecnologías bien asentadas como Ethernet o Infiniband. Existen en cambio, tecnologías de transmisión a larga distancia más avanzadas, donde un único enlace de fibra óptica puede transmitir más de 32 Tbit/s [9].

1.1. Motivación

Mantener una infraestructura de red de las características previamente mencionadas, requiere de una elevada inversion de capital inicial o Capital expenditures (CAPEX) y de una capital para las operaciones u Operational Expenditures (OPEX). Garantizar el buen funcionamiento de esta infraestructura es necesario, independientemente del tamaño o de la velocidad de la red, para poder rentabilizar la inversión. Es bajo este contexto en donde aparece la necesidad de una buena *gestión de red*.

La *gestión de red* implica la toma de decisiones de todo tipo, como la planificación y actualización del equipamiento hardware, o las reglas de enrutado del tráfico. Para poder tomar decisiones adecuadas, es imprescindible disponer de la máxima información posible en todo momento. Es por este motivo, que la *monitorización de la red* juega un papel crítico a la hora de realizar una gestión de calidad. No obstante, la dificultad de monitorizar una red crece tanto con la complejidad de la propia red (número de puntos de monitorización necesarios), como con la velocidad de los enlaces, motivo por el cual la monitorización de red se ha convertido en un problema recurrente a lo largo de los años [5–7].

Para entender el impacto que tiene sobre la monitorización de red la velocidad de los enlaces, por ejemplo, en un centro de datos [8], es necesario entender el volumen y su coste de cómputo y análisis. Supongamos un enlace de 100 Gbit por segundo. En el peor caso, en donde todos los paquetes tendrán tamaño 60 Bytes, se transmitirán 148.8 millones de paquetes por segundo, quedando por tanto un tiempo entre paquetes de tan solo 6.7 nanosegundos. Para entender lo pequeño que es este número, un procesador moderno como el Intel Skylake a 3.4 GHz, presenta una latencia de lectura de ~3.5 ns (en el mejor caso) sobre una cache de nivel 2 y ~12.9 ns sobre una cache de nivel 3 [10]. Dado que una línea de caché son 64 Bytes, leer un paquete de la caché de nivel 3 y moverlo a la caché de nivel 1 lleva más tiempo que la transferencia del propio paquete por la red, imposibilitando por tanto realizar cualquier tipo de procesamiento utilizando un único hilo de ejecución. Por este motivo, es posible afirmar que la paralelización es una parte fundamental del proceso de monitorización en redes de 10 Gbit/s y superiores.

Adicionalmente, debido al volumen de tráfico, es importante entender que el proceso de monitorización no puede reducirse únicamente a obtener información del tráfico de la red. También es fundamental entender esta información, y con-

densarla en elementos interpretables por un operador [11]. Recordemos que en una red de 100 Gbit/s podemos llegar a transportar hasta 148.8 millones de paquetes por segundo, lo que da una idea de la complejidad. Por este motivo, el proceso de monitorización debe paralelizarse, no solo a nivel de hilo en la fase de captura [12], sino también al repartir el cómputo entre diferentes equipos [13].

El objetivo de esta tesis es proponer una solución de monitorización integral, que no solo permita realizar una captura y distribución de los datos, sino que ofrezca capacidades de análisis avanzando para tener un conocimiento profundo del estado de la red, de modo que esta información sea útil para la actuación del gestor de red. Para lograrlo, se define una solución de monitorización integral como una aplicación dividida en 4 componentes bien diferenciados (ver Figura 1.2): (I) Captura de datos de la red, (II) Procesamiento y distribución, (III) Análisis, y (IV) visualización.

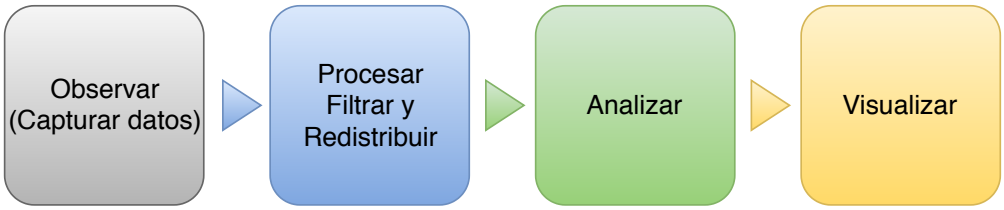


Figura 1.2. Arquitectura propuesta para una monitorización integral

1.2. Estructura de la Tesis

Tal y como se ha expuesto previamente, una de las partes más complejas de la monitorización es la captura de los datos, ya que las soluciones son dependientes de la tecnología red. En el Capítulo 2, se expone el estado del arte de las diferentes técnicas de monitorización hasta la actualidad, y cuáles son las propuestas para abarcar los problemas de las redes 10, 40 y 100 Gbit Ethernet, tanto a nivel de monitorización activa como pasiva.

Una vez se dispone de las herramientas de captura, es necesario preprocesar la información de la red y minimizar el volumen de datos a analizar. Este proceso se aborda en el Capítulo 3, en donde tras evaluar diferentes soluciones Big Data para la distribución de datos, se decide desarrollar una solución orientada al reparto y preprocesado de datos, recibidos a alta velocidad, como es el caso de las redes de 100 Gbit/s.

Una vez los datos han sido correctamente distribuidos, preprocesados y filtrados, es necesario realizar un análisis de los mismos, con el objetivo de extraer información útil. El Capítulo 4 presenta algunos de los análisis que se han realizado, de modo que se puede apreciar fácilmente las capacidades que ofrece el sistema de monitorización Big Data propuesto. En este capítulo también se muestran algunos métodos de visualización de los datos analizados y como estos métodos facilitan a los gestores de red su trabajo.

Finalmente, en el Capítulo 5 se exponen las conclusiones y trabajo futuro para cada una de las fases en las que se divide la arquitectura Big Data propuesta, así como para ampliar las capacidades de monitorización, análisis y seguridad de las redes de comunicaciones.

2

Muestreo de la red

La monitorización de red se puede definir como un proceso de vigilancia sobre los componentes que forman la red. Para realizar este proceso es necesario obtener métricas de estos dispositivos. El muestreo de datos se puede realizar de forma activa o pasiva, presentado cada estrategia sus propias ventajas e inconvenientes [14]. La monitorización activa más simple consiste en consultar a todos o a un subconjunto de los dispositivos monitorizables por sus estadísticas de utilización. El tipo de información que un componente de red puede almacenar es muy variado. No obstante, el volumen de datos a almacenar, así como el cómputo necesario para registrarlos, es proporcional al número de puertos y el volumen de tráfico que atraviesa al dispositivo de red. Dado que los procesos de monitorización son una funcionalidad secundaria frente a la propia gestión del tráfico, los recursos de computación y memoria reservados para este objetivo son reducidos. La información más básica que puede almacenar un dispositivo de red es la de bajo nivel, cómo la conectividad de los enlaces, la potencia de la óptica, la saturación de las colas de transmisión y recepción, el uso de CPU o la memoria, etc. Algunos dispositivos son capaces de almacenar información un poco más detallada del tipo de tráfico en curso, cómo las direcciones IP y puertos

de flujos activos que están atravesando el dispositivo en un momento determinado y alguna característica básica sobre los mismos.

Uno de los puntos negativos de este tipo de monitorización activa es la falta de homogeneidad en el equipamiento utilizado. La falta de homogeneidad en la monitorización realizada por los equipamientos de red genera problemas para obtener resultados Es común observar equipamiento de diferentes fabricantes –y en algunas ocasiones diferentes generaciones dentro del mismo fabricante– no compartan ciertas funcionalidades, protocolo de gestión o algunos conjuntos de estadísticas. En 1988 nació el Simple Network Management Protocol (SNMP) [15] cuyas siglas se pueden traducir por *protocolo de gestión de red simple*. Dicho protocolo pretendía homogeneizar la gestión y la monitorización del equipamiento de red proporcionando, cómo su nombre indica, un método simple. No obstante, y a pesar de que este protocolo se encuentra disponible en la mayoría –sino en todo– el equipamiento de red actual, algunos consideran al protocolo SNMP un fracaso lleno de fallos de seguridad [16], rendimiento [17] e incompatibilidades.

Debido a la propia flexibilidad del protocolo, SNMP permite definir las denominadas Management Information Bases (MIBs), que podrían definirse como pequeños grupos de variables y funciones que permiten gestionar un conjunto de funcionalidades del dispositivo o estadísticas. Esta misma flexibilidad causa que cada fabricante utilice sus propias MIBs propietarias, creando una heterogeneidad incluso entre sus propios productos. Esto implica la necesidad de una herramienta parcialmente diseñada a medida del equipamiento de red que se esté utilizando, diseño que deberá actualizarse probablemente con la incorporación de nuevo equipamiento. Esto también ha fomentado que algunos fabricantes usasen estas restricciones para vender sus herramientas de gestión y monitorización que únicamente eran compatibles con sus productos en un claro intento de monopolizar centros de datos.

Otro método de aplicar la monitorización activa parte de la realización de pruebas punto a punto dentro de la red. Esta técnica permite obtener métricas puntuales difíciles de obtener con otros métodos, como la latencia [18], el jitter [19], o el ancho de banda disponible [20]. Este tipo de medidas son fundamentales a la hora de servir o transportar servicios multimedia por la red [19, 21]. Como punto negativo, la inyección de tráfico en la red afecta al estado de la misma produciendo posibles mediciones erróneas o incluso presentar un peligro de efectuarse en una red saturada [22].

Los métodos pasivos de monitorización son más sencillos a nivel conceptual, dado que su único objetivo se centra en observar el tráfico que circula por determinados enlaces de la red y realizar estimaciones de su estado a partir de los datos obtenidos. La monitorización pasiva, por tanto, no tiene ningún impacto negativo en la medida, pues no introduce ningún esfuerzo computacional al equipamiento de red al no inyecta ningún tipo de tráfico que perturbe su funcionamiento. Una de las ventajas de este tipo de mediciones, es la posibilidad de analizar de forma no intrusiva el funcionamiento de las aplicaciones que hacen uso de la red, como servidores web, bases de datos y por supuesto, verificar la seguridad y accesos de estos sistemas. No obstante, la monitorización pasiva es el método de monitorización más costoso a nivel computacional y el que más recursos hardware necesita, tal y como se detalla en la Sección 2.1.

Los métodos de evaluación de red, pasivos y activos, tienen cada uno su aplicabilidad y sus escenarios de uso. Por este motivo, a la hora de proponer un sistema de monitorización completo e integral, es necesario disponer de las herramientas adecuadas para obtener métricas tanto activas como pasivas, y poder escoger la mejor en el momento adecuado. A lo largo de este capítulo se exploran ambas aproximaciones, y se proponen nuevas soluciones para cada método.

2.1. Monitorización Pasiva

Los métodos pasivos de monitorización han sido comúnmente los más utilizados y explotados tanto en el ámbito académico como por las grandes empresas e ISPs. No obstante, debido a la complejidad que este tipo de monitorización conlleva, se han desarrollado multitud de métodos y estrategias para obtener información útil en un formato manejable, tanto a escala computacional como humana. El primer paso para reducir el volumen de datos a analizar fue pasar del concepto de paquete individual, al concepto de flujo de red [6]. Una posible definición de flujo de red es: una conexión punto a punto entre dos elementos conectados a la red con una cierta entidad propia. No obstante, esta definición es muy ambigua, permitiendo varias interpretaciones posibles. La definición más común se basa en el concepto de quíntupla [23]. La quíntupla está formada por 2 direcciones de red (típicamente direcciones IP), 2 puertos a nivel de transporte y el protocolo de nivel de transporte utilizado (normalmente TCP o UDP).

Los primeros sistemas de monitorización que empezaron a utilizar el concepto de flujo fueron los propios routers con el protocolo Netflow desarrollado por Cisco [24]. La idea de este protocolo era sencilla: ya que los routers observan de forma natural el tráfico que circula por la red, el coste de anotar de forma breve en formato de quíntupla, que paquetes atravesaban y la red y en qué dirección, no era especialmente elevado. Al cabo de cierto tiempo sin que el router transfiriese ningún paquete perteneciente a un flujo, o en su defecto se observase un paquete que indicase que la conexión se cerraba, el equipamiento “expiraba” el flujo. Este proceso consistía no solo en liberar la memoria con la estadística relacionada con el flujo, sino que el router transmitía un paquete de Netflow a un equipo previamente configurado con la estadística almacenada.

Inicialmente, la estadística almacenada por los equipos de red era escasa, pero con la evolución de las capacidades de cómputo e incremento de memoria estos equipos, se empezó a almacenar más y más información, haciendo evolucionar el protocolo Netflow hasta la versión 9 [25], e incluso desarrollar un nuevo protocolo mucho más completo y dinámico: IPFIX [26, 27]. No obstante, la evolución de la tecnología no solo influyó positivamente en los protocolos de monitorización del equipamiento de red, sino que en paralelo los requisitos computacionales y de memoria crecían rápidamente con el propio crecimiento de la red, haciendo más y más complicado mantener el estado de los flujos por estos equipos.

El coste computacional de enrutar millones de paquetes por segundo no es despreciable. Esto complica la monitorización, pues a pesar de ser un proceso crítico en una red, si el equipamiento de red no tiene recursos suficientes para realizar la monitorización, cometido principal para el que fue diseñado, deberá relegar la monitorización a un segundo plano. Bajo estas limitaciones aparecieron las técnicas de muestreo (*sampling*) de paquetes [28], es decir, aplicar una regla de bajo coste para escoger 1 paquete entre N y excluir el resto del proceso de monitorización. La idea del muestreo es sencilla conceptualmente, pero presenta ciertos inconvenientes no triviales de solucionar.

Existen multitud de experimentos y trabajos que han analizado y propuesto diferentes métodos y criterios para la realización del muestreo. En muchos casos, estos trabajos permiten obtener con un mayor o menor coste computacional una vista general del estado de la red. No obstante, esto puede no ser suficiente [29], ya que los flujos de pocos paquetes podrían llegar a pasar completamente desapercibidos [30]. La idea de crear un método de muestreo que no se base en la estadística y deba analizar el paquete en sí, permitiría evitar muchos de los problemas del muestreo. Como inconveniente, el coste de este método sería comparable o superior a realizar el propio análisis que el equipamiento de red no puede permitirse, impidiendo que cualquier método de sampling que requiera consultar campos internos de los paquetes sea rentable computacionalmente. Desde el punto de vista de la seguridad esto es un problema, por ejemplo, existen ataques de denegación de servicio con un bajo ancho de banda [31].

Para poder tener una visión del estado de la red más fino, es necesario disponer de equipos especializados para realizar la monitorización y análisis de la red. Una de las posibles soluciones es recurrir al hardware específico como los cicuitos a medida o Application-Specific Integrated Circuits (ASICs) o recurrir a hardware reconfigurables cómo las Field Programmable Gate Arrays (FPGAs) [32–34], no obstante, este tipo de sistemas tienen un coste elevado y son difíciles de mantener y escalar. Además, su elevada complejidad causa que la diferencia entre una tecnología de red y su correspondiente herramienta de monitorización en hardware, se retrase fácilmente años. Por este motivo es necesario recurrir a soluciones desarrolladas en software, ya que son más dinámicas y versátiles.

2.1.1. Herramientas de procesamiento de tráfico a 10Gbit/s

A la hora de pensar en la captura de tráfico en software, es necesario recordar la magnitud de este problema en el caso peor. En un enlace Ethernet de 10 Gbit/s, el cual está ampliamente establecido a día de hoy en centros de datos, la tasa máxima de paquetes por segundo puede alcanzar los 14.88 Millones de paquetes. No obstante, si bien los 10 Gbit/s pueden ser la norma de los enlaces entre equipos, es común ver enlaces agregados entre equipos de red de 40 o 100 Gbit/s, pudiendo alcanzar hasta 148.8 millones de paquetes por segundo en este último caso. Dentro de un sistema operativo, cada paquete debe superar la pila de red hasta alcanzar el nivel de usuario y aplicación. A nivel de sistema operativo, podemos definir la pila de red como toda aquella funcionalidad que obtiene el paquete desde la tarjeta de red, lo analiza verificando su integridad y su destinatario, y finalmente si procede, lo agrupa como un flujo y lo entrega a la aplicación de nivel de usuario oportuna. Esto también se aplica a los sistemas de monitorización, ya que, aunque no es necesario que cada paquete atraviese la pila al completo, sigue existiendo cierto sobrecoste. A este hecho, hay que añadirle que la pila del Kernel de Linux no es paralelizable, por lo que en todo caso los casi 15 o casi 150 millones de paquetes deberían ser capturados por un solo hilo de ejecución, y como vimos anteriormente es una tarea prácticamente imposible con los procesadores actuales.

Para solventar este problema, se han desarrollado multitud de herramientas para facilitar el trabajo al sistema operativo y al software de captura y monitorización. Una de estas aproximaciones, pasa por asumir que el propio software no será capaz de hacer la mayor parte del análisis, por lo que es necesario hardware especializado que filtre la información previamente. Algunos ejemplos de este tipo de hardware son las tarjetas fabricadas por Tilera [35]. Su arquitectura consta de un grupo de procesadores programables desde el Host que permiten hacer filtrado, inspección profunda y multitud de otras opciones interesantes. No obstante, estas tarjetas tienen un mercado muy pequeño, por lo que su precio es muy elevado. Otro inconveniente de estas tarjetas es la memoria local. Obtener información a nivel de flujo requiere no solo capturar los paquetes a la tasa del enlace, sino llevar una estadística de los mismos. Esto incrementa rápidamente la memoria necesaria por cada flujo y contador que deseemos utilizar. Una opción parecida a las tarjetas Tilera son las tarjetas de red basadas en FPGA como la NetFPGA [33]. Estas tarjetas ofrecen una mayor flexibilidad que una tarjeta Tilera, pero comparativamente el tiempo de desarrollo es varias veces superior por un

coste monetario equivalente. Estos dispositivos, tampoco son capaces de disponer de una memoria capaz de manejar con soltura la construcción de flujos en enlaces multigigabit.

La última opción para minimizar los costes, y poder utilizar grandes cantidades de memoria para la construcción de flujos, es la utilización de hardware de propósito general. No obstante, ya que este hardware está idealmente diseñado para ser utilizado a nivel de usuario, sus drivers hacen uso de la pila de red del sistema operativo, y ya se ha explicado que no es una solución adecuada. Es por ello que esta opción requiere desarrollar software específico, capaz de evitar o minimizar al máximo el impacto de la pila de red, las propias interrupciones de la tarjeta de red, y otros eventos de bajo nivel que no son necesarios para este caso de uso. Existen multitud de herramientas o *frameworks* que cumplen estas características, y que has sido desarrollados tanto por empresas como por la comunidad académica, para evitar al máximo los overheads impuestos por los drivers *legacy* o *vanilla*. A continuación se muestra un poco el funcionamiento de las herramientas más populares.

PacketShader

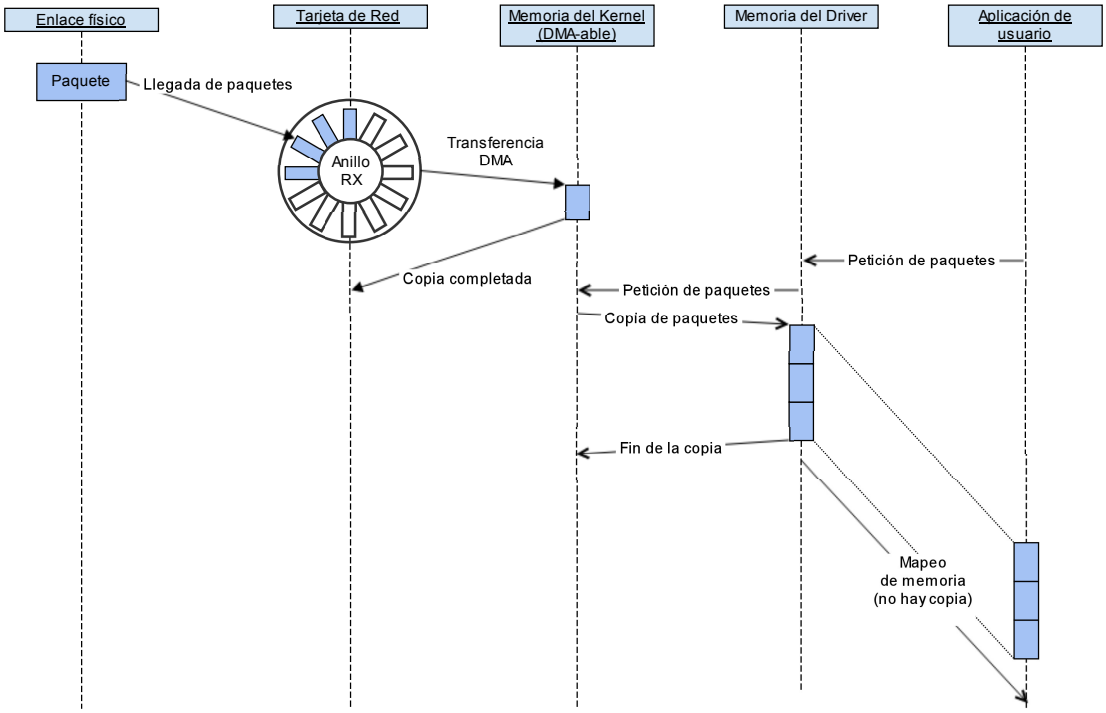


Figura 2.1. Flujo de paquetes en PacketShader

PacketShader [36] fue uno de los primeros motores de captura y gestión de paquetes en implementar un driver de gestión propio. Además, también fuel el primero es hacer uso de coprocesadores (más concretamente coprocesadores gráficos: Graphics Processing Units (GPUs)) para la fase de análisis del tráfico capturado. El objetivo inicial de *PacketShader* fue el de construir un router a alta velocidad. No obstante, su principal aportación radica en que sus ideas, todas ellas muy novedosas en su momento, han sido el punto de partida de los motores de captura "software", como veremos a lo largo de esta sección.

A nivel conceptual, una de las mayores aportaciones del diseño de *PacketShader* fue el uso del denominado *onecopy* (una copia). El término *onecopy* viene del inglés *una copia*, y hace referencia a como se gestionan los paquetes internamente dentro del driver de red. El paquete, una vez ha sido copiado a una región de memoria específica del driver, se copia a una segunda región dentro del driver, que será expuesta al usuario. Esta copia es necesaria para asegurar que errores a nivel de usuario no puedan corromper memoria relacionada directamente con el hardware y el Kernel, permitiendo, por ejemplo, leer memoria de otros usuarios o

bloquear el host. Esta memoria a su vez es *dmeable*, es decir, lista para poder ser transferida directamente a una GPU, motivo extra para realizar esta copia. Este flujo de datos, ha sido representado en la Figura 2.1

El método de realizar copias intermedias de paquetes, a pesar de presentar ciertas ventajas a nivel de seguridad o para un uso específico del usuario, tiene un importante efecto en el rendimiento final debido a dos factores: (I) la propia limitación del ancho de banda de la memoria, que debe ser escrita y leída 2 veces por paquete antes de haber iniciado cualquier proceso de análisis y (II) la copia debe ser realizada por la CPU, perdiendo valiosos ciclos que podrían ser utilizados por un análisis posterior o en la captura y gestión de otro paquete. Esto lleva al driver *PacketShader* a tener dificultades a tasas cercanas y superiores a los 10 Gbit/s.

PacketShader también fue el primero en aplicar el alineamiento de paquetes a líneas de caché. Esto hace que una línea de caché no se comparta entre dos paquetes que potencialmente podrían procesarse en paralelo. Además, normalmente la mayor parte de la información útil se encuentra en los primeros bytes del paquete, así que un análisis sencillo solo utilizaría una línea de caché independientemente del tamaño del paquete. Como parte negativa, *PacketShader* solo solicita paquetes a la tarjeta de red cuando la aplicación de usuario lo solicita. Esto causa un pequeño cuello de botella, ya que la cola interna de la tarjeta es pequeña, y si la aplicación de usuario no es suficientemente rápida, la probabilidad de perder paquetes es muy elevada. Otro dato a tener en cuenta de *PacketShader* es que carece de una Application Programming Interface (API) estándar, por lo que todas las aplicaciones que hagan uso de ella deberán ser reescritas parcialmente.

PF_Ring DNA

PF_Ring [37, 38] probablemente fue el primer motor de captura. Diseñado para tarjetas Intel de 1 y 10 Gbps, este *framework* fue presentado por primera vez el mismo año en el que AMD presentó los primeros procesadores con dos núcleos. Esto, hizo que *PF_Ring* fuese evolucionando rápidamente convirtiéndose en uno de los primeros sistemas de captura en, además, explotar el paralelismo en el proceso de captura. No obstante, esta mejora tuvo que esperar hasta que las tarjetas de red incorporaron las denominadas colas Receive Side Scaling (RSS) (receive side scaling) [39], que permiten dividir el tráfico recibido entre diferentes núcleos (cores) y CPUs en función de un determinado hash. Esta tecnología, introducía

muchas mejoras a bajo nivel, ya que esta división incluía colas independientes, cada una asociada a sus propias interrupciones.

A diferencia de *PacketShader*, *PF_Ring* decidió optar por la técnica de *zerocopy* en el tratamiento de paquetes. Como su nombre indica, un procesamiento *zerocopy* debe tener cero copias de paquetes a nivel de Kernel o, dicho de otro modo, la misma región de memoria en donde la tarjeta de red copia los paquetes mediante el uso de accesos directos a memoria o Direct Memory Access (DMA), es expuesta directamente a las aplicaciones de usuario. Esto es potencialmente un problema de seguridad pero a cambio proporciona un rendimiento superior. Es interesante destacar que *PF_Ring* proporciona una API amigable con el usuario ya que cumple el formato estándar de facto: *libPCAP*. En la Figura 2.2 se muestra el flujo de datos que siguen los paquetes hasta llegar a una aplicación de usuario.

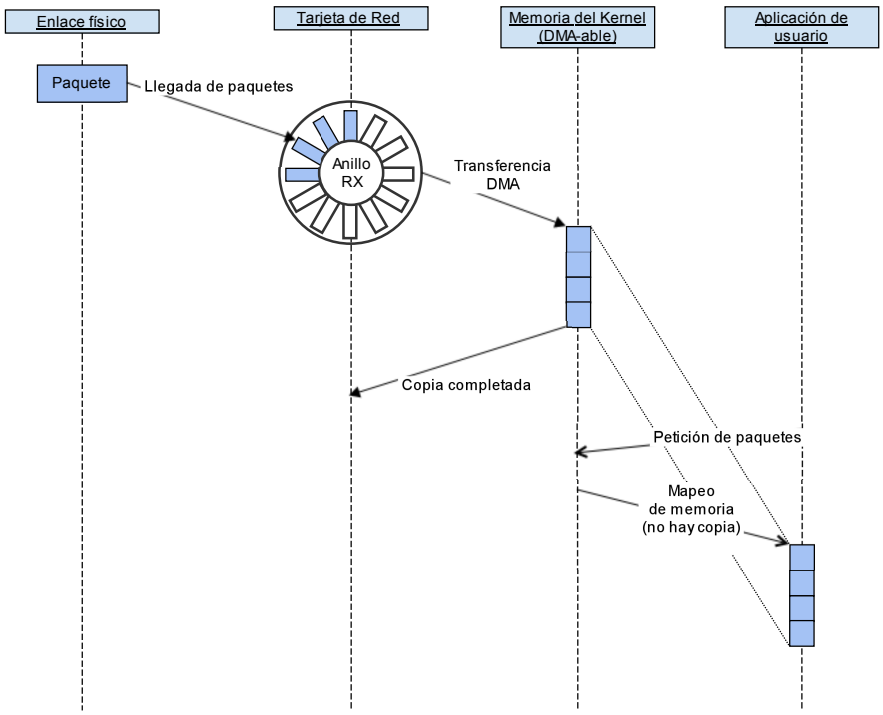


Figura 2.2. Flujo de paquetes en PF_Ring

Netmap

Netmap [40] es uno de los motores de captura más conocidos. Este sistema de captura se diseñó utilizando ideas de motores contemporáneos [41] como *PacketShader* o *PF_RING*. Netmap decidió utilizar la gestión de paquetes *zerocopy*,

a la vez que mantiene de forma asíncrona utilizando *polling* sobre bloques de paquetes, mejorando el funcionamiento en bloques que realiza *PacketShader* en sus copias intermedias. Netmap a diferencia de *PF_RING* centra parte del esfuerzo en la seguridad en las regiones de memoria expuestas al usuario, minimizando las posibilidades de corrupción de memoria del Kernel por parte de la aplicación, ya sea deliberada o por un error del programador. Además, Netmap, a diferencia de los anteriores, no se presenta como driver exclusivo para tarjetas Intel, sino que se integra con otros fabricantes como Realtek o Nvidia. A nivel de API ofrece una interfaz de sockets clásica, pudiendo usar si se desea *libPCAP* u otra solución a menor nivel. El flujo de datos del paquete es equivalente al de *PF_RING* (ver Figura 2.2).

HPCAP

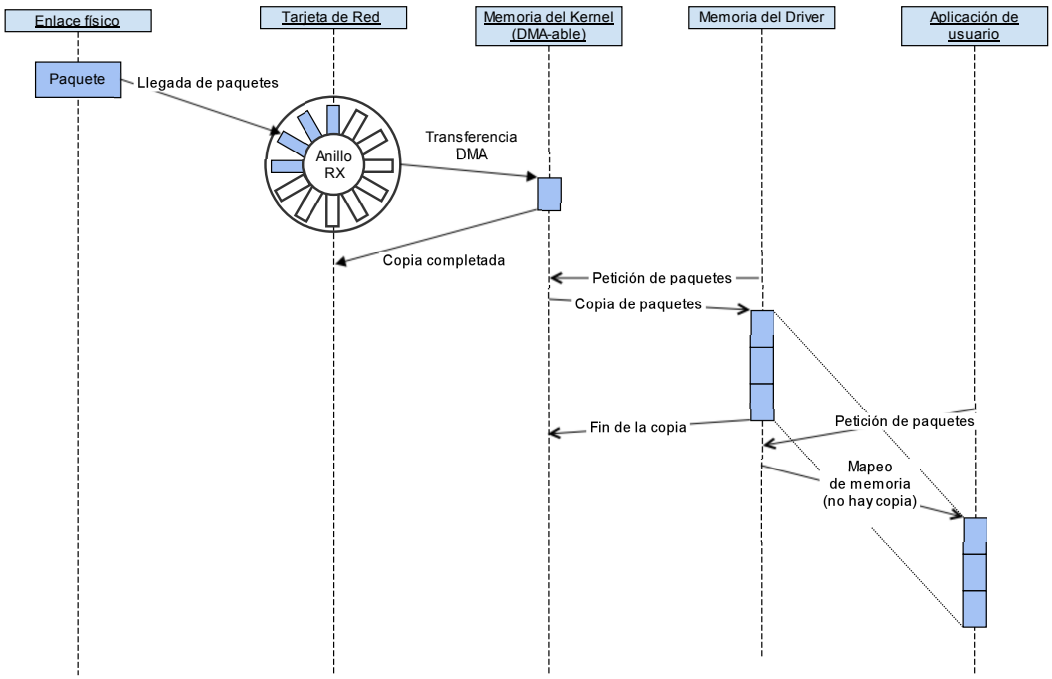


Figura 2.3. Flujo de paquetes en Hpcap

El driver *HPCAP* a 10 Gbit/s fue desarrollado en el grupo de investigación HPCN [42]. Este driver, es una extensión mejorada del driver oficial de las tarjetas Intel de 10 Gbps, de forma similar a *PacketShader* o *PF_RING*, por lo que igualmente es solo es compatible con ciertos modelos de este fabricante. Al ser una

extensión, este motor de captura conserva en gran medida parte de la funcionalidad original del driver, permitiendo a las interfaces de red funcionar en modo *vanilla* o en modo *HPCAP* indistintamente. Al contrario que los otros motores de captura presentados anteriormente, *HPCAP* tiene la capacidad de capturar la mayor parte del tráfico con una única cola de recepción gracias a su eficiencia. No obstante, puede hacer uso de las colas RSS al igual que cualquier otro motor de captura moderno.

Al igual que *PacketShader*, *HPCAP* se basa en la filosofía de *onecopy*. La razón de utilizar esta filosofía es la misma que en *PacketShader*: agrupar los paquetes en una región contigua y bien alineada (o buffer). Esto permite a HPCAP poder hacer de forma relativamente trivial escrituras a disco a tasa de línea, algo que para un sistema *zerocopy* requeriría realizar esta copia intermedia igualmente a nivel de usuario, pudiendo en este caso ser incluso más ineficiente. Otra ventaja de este buffer es su tamaño, típicamente 1 Gigabyte, que permite a las aplicaciones con un retardo por paquete variable, contar con un buffer lo suficientemente grande como para compensar ráfagas de tráfico. La API de acceso a HPCAP es similar a la del estándar *libPCAP*, pero no seguirlo de forma estricta requiere reescribir las aplicaciones de monitorización para poder utilizarlo. En la figura 2.3 se muestra el flujo de datos que siguen los paquetes hasta llegar a una aplicación de usuario.

Data Plane Development Kit

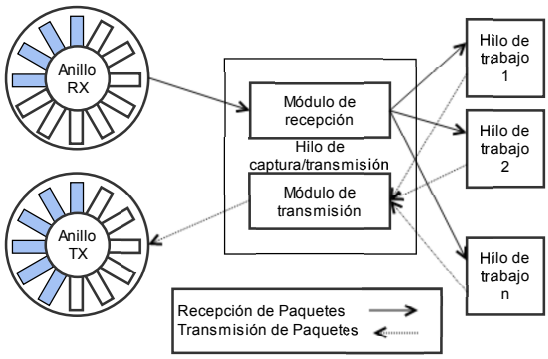


Figura 2.4. Flujo de paquetes en una aplicación Intel DPDK

El Data Plane Development Kit (DPDK) [43], también conocido en sus inicios cómo Intel DPDK, se dio a conocer públicamente en torno a 2014 de la mano de Intel. No obstante, Intel acabó abandonando el proyecto dejándolo en manos de

la comunidad de software libre, y permitiendo la libre entrada de desarrolladores y fabricantes. Gracias a esto, DPDK soporta todo tipo de tarjetas de un conjunto muy amplio de fabricantes y dispositivos[1]. Gracias a esto se ha convertido en el marco de referencia en los sistemas de procesamiento de paquetes, y una pieza indispensable de lo que hoy conocemos como Software Defined Networking (SDN) y Network Function Virtualization (NFV) [44].

DPDK representa una filosofía de cómo se debe diseñar y ejecutar una aplicación orientada a la red. Esta filosofía no es realmente nueva, pues nace como una recopilación de las buenas prácticas de todos y cada uno de los motores de paquetes anteriores a él. DPDK plantea un sistema multihilo y multicore, en donde cada componente (recepción, análisis, transmisión, etc.) se ejecuta en uno o más hilos asignados a cores de CPU aislados, y su comunicación con el resto de componentes se realiza mediante colas circulares de paquetes 2.4.

A nivel técnico DPDK presenta una abstracción del hardware y del Kernel muy interesante y novedosa. DPDK busca eliminar al máximo la interacción con el Kernel y, por ende, los problemáticos cambios de contexto entre espacio de usuario y Kernel. Para lograrlo recurre a los módulos *UIO* y *VFIO*. Ambos módulos, aunque con sus restricciones y particularidades, permiten exponer dispositivos hardware (a nivel de registro y memoria) a las aplicaciones de usuario. De esta forma, es posible programar drivers en un espacio protegido de usuario, lo que implica poder utilizar funciones básicas que no están disponibles de forma sencilla a nivel Kernel (como accesos a ficheros o interactividad con otras aplicaciones), con la seguridad de que un acceso no autorizado causará problemas en exclusiva al proceso en ejecución y no al sistema completo, como en otros motores de paquetes más integrados a nivel de Kernel.

Ya que gran parte del desarrollo de esta tesis se ha realizado sobre DPDK, es preciso introducir el funcionamiento interno. El concepto de paquete en DPDK es denominado *mbuf* y corresponde al clásico *packet-descriptor* usado en la pila de red del Kernel de Linux. Un *mbuf* es una estructura que contiene cierta información acerca del propio paquete, como en que puerto fue recibido, su longitud, etc. Un *mbuf* también porta un puntero a donde está ubicado el contenido en binario del paquete. Esto es una solución interesante, ya que permite implementar la filosofía *zerocopy* de forma directa, es decir, al copiar, encolar o desencolar un *mbuf*, en ningún caso se realiza una copia de su contenido, y esto también se apli-

[1] `http://dpdk.org/doc/nics`

ca a la tarjeta de red, que desde un primer momento lee y escribe en las regiones descritas por el *mbuf*.

Drivers Mellanox

La compañía Mellanox es la principal potenciadora de la tecnología de transmisión *Infiniband*. Tradicionalmente, esta tecnología tiene mejores prestaciones que Ethernet, ya que permite velocidades de transferencia muy superiores y una latencia entre mensajes muy inferior. No obstante, debido a su elevado coste y aplicabilidad casi en exclusiva a sistemas de supercomputación, los drivers y las APIs de acceso de estas tarjetas de red han sido olvidadas como parte del estado del arte de los sistemas de monitorización. Sin embargo, Mellanox también comercializa a día de hoy tarjetas Ethernet, que hacen uso de sus drivers con una API similar a la que ofrecían con *Infiniband*. Esto permite a una aplicación de usuario acceder a nivel de paquete y registro de la tarjeta, pudiendo realizar aplicaciones de monitorización de muy alto rendimiento (40 o 100 Gbit/s!) sobre la tarjeta, sin necesidad de recurrir a drivers o *frameworks*. Un ejemplo de esto es DPDK, cuyo soporte de tarjetas Mellanox consiste en añadir un pequeño envoltorio a la funcionalidad que ya proporcionan los drivers originales.

A pesar de todo, estos drivers son propietarios, y sus funciones son complejas y nada amigables. Sumado al hecho de que solo funcionan sobre tarjetas Mellanox, y que este fabricante tenía una cota de mercado reducida previo a DPDK, hizo que estas APIs nunca llegasen a estandarizarse y solo fuesen usadas por aplicaciones muy concretas, como el núcleo de MPI.

2.1.2. Virtualización en redes de comunicaciones

Las futuras tecnologías móviles 5G van a ser autónomas [45] y segmentadas [46]. En este contexto, estas redes tienen que implementar el ciclo Observar, Analizar y Actuar (OAA) [47] que les permita ser auto gestionadas con una intervención humana mínima. El proceso de observación debe comprobar la salud de la red regularmente, monitorizando cada uno de los elementos que la componen, así como el tráfico que los atraviesa. Esta monitorización puede realizarse tanto de forma activa como de forma pasiva. La monitorización activa puede resultar peligrosa en situaciones de tráfico elevado. En cambio, los métodos pasivos proveen una imagen exacta del estado de la red en todo momento, incluso cuando se encuentra fuertemente cargada, situación de máximo interés desde un punto de vista de gestión de red.

Según la ITU [48], las futuras redes 5G deberán tener al menos 20 Gbit/s de bajada y 10 Gbit/s de subida por cada estación base. Por tanto, la monitorización pasiva del tráfico en estos nodos requerirá al menos interfaces de captura de 40 Gbit/s –en caso de utilizar el estándar Ethernet en dichas estaciones– para poder alimentar al ciclo OAA. Aunque esto es un gran reto, la monitorización a 40 Gbit/s es solo un paso para acercarse a sistemas de monitorización de velocidades superiores como los 100 o 400 Gbit/s Ethernet [49].

Adicionalmente, la segmentación de las redes 5G implican la virtualización de recursos mediante el uso de conceptos como SDN o el NFV [44] En este contexto, el *Broadband Forum* ha propuesto una arquitectura para la virtualización denominada *Cloud Central Office* (CloudCO) [50]. En esta aproximación, la mayor parte de los elementos de red se encuentran encapsulados en los denominados *Macro-Nodes*, que virtualizan todos los elementos necesarios. Esta propuesta, al igual que otras como la que se está desarrollando en el proyecto europeo (H2020) Metro-Haul[2], tiende a reducir el sobre-aprovisionamiento y la intervención humana. Esto permite reducir los costes tanto en CAPEX como en OPEX, lo cual es deseable para cualquier ISP.

En este punto, parece obvio que para monitorizar las redes 5G será necesario, no solo superar la barrera de la captura a 40 Gbit/s sino utilizar nodos virtualizados. En este escenario, se propone el concepto de sonda de red virtualizada (Virtual Network Probe (VNP)), como aquel software ejecutado en una máquina virtual encargado de la monitorización de tráfico tanto proveniente de una inter-

[2] `https://metro-haul.eu/`

faz física, como de la comunicación entre máquinas virtuales. El término VNP también representa un caso de uso de la tendencia llamada *Monitorización bajo servicio* (*Monitoring-as-a-Service* (MaaS)) [51, 52]. Además de todas las ventajas proporcionadas por una NFV, la VNP puede resolver algunos de los nuevos problemas de monitorización introducidos por la virtualización, que no se encuentran en los escenarios clásicos de máquinas completamente dedicadas.

Hasta ahora, los despliegues de monitorización han tenido una arquitectura muy clara y lineal: El hardware de monitorización es conectado a un puerto de *mirroring* del switch por el que circula el tráfico de interés. Este switch copia los paquetes de aquellos puertos que se desean monitorizar a la interfaz de *mirroring*, y el hardware de monitorización lo captura y analiza. Cuando la virtualización entra en juego, existen otras configuraciones que complican o imposibilitan esta configuración de monitorización.

Un primer ejemplo, que no requiere que toda la infraestructura se encuentre virtualizada, es el caso de los entornos distribuidos con un elevado número de servidores y posibles puntos de monitorización. En esta situación, desplegar un conjunto de sondas físicas por cada punto de monitorización puede ser caro. Sin embargo, si la infraestructura provee nodos de cómputo virtualizados, el coste de desplegar VNPs sobre estos equipos es prácticamente cero. El tráfico puede ser redirigido externamente a los nodos de cómputo donde se ejecutan las VNPs, monitorizando bajo demanda sin necesidad de intervenir o modificar la infraestructura física de la red.

Esta aproximación se puede extender a situaciones donde la infraestructura está completamente virtualizada, haciendo uso de NFVs, donde el tráfico a monitorizar se transmite entre diferentes máquinas virtuales a través de funciones virtuales (Virtual Functions (VFs)). En algunas situaciones, los datos transferidos no abandonan el servidor físico en donde se encuentran alojadas las máquinas virtuales, y por ende, ninguna sonda física podría monitorizar este tráfico salvo que fuese copiado deliberadamente al exterior, proceso que dependiendo del hypervisor y de la infraestructura virtual podría no ser posible. En cambio, las funciones virtuales pueden ser configuradas en modo espejo (*mirror*) al igual que un switch, permitiendo a un VNP entrar en juego y analizar ese tráfico.

Ambos escenarios son interesantes para las redes móviles 5G, ya que cómo se ha descrito anteriormente, las redes 5G utilizará una arquitectura virtualizada. La aproximación de utilizar VNPs en estos escenarios permite a los operadores de red, tanto de forma manual como automática, monitorizar tráfico tanto en enla-

ces virtuales como físicos sin necesidad de intervención humana en los sistemas físicos, reduciendo costes y aumentando la velocidad de reacción ante eventos o cambios en la red.

Como se ha explicado con anterioridad, el proceso de captura tiene unos elevados requisitos computaciones y esto conlleva un problema al ejecutarse en un entorno virtualizado [53]. Aunque a día de hoy, sistemas de virtualización modernos tienen un impacto reducido en el rendimiento, es necesario evaluar la tecnología, ya que el tiempo de transmisión de un paquete de tamaño mínimo a 40 Gbit/s es de apenas 16.8 nanosegundos.

Tecnologías de virtualización

Actualmente, a pesar de los sobrecostes, la virtualización no es considerada una opción de bajo rendimiento. Algunos hypervisores como Linux Kernel Virtual Machine (KVM) o XEN utilizan funcionalidades hardware que minimizan la intervención por parte del propio hypervisor. Esta tecnología se encuentra presente en la mayoría de los procesadores actuales y recibe un nombre específico en función del fabricante, por ejemplo, VT-d es utilizado en los procesadores Intel y Vi en los procesadores de AMD. Este hardware especial permite virtualizar componentes físicos introduciendo nuevas posibilidades. La tecnología de *PCI passthrough*, por ejemplo, permite asignar un dispositivo PCI físico a una máquina virtual, dándola control completo a dicho dispositivo PCI a la vez que se asegura un aislamiento seguro entre anfitrión y máquina virtual. A pesar de que *PCI passthrough* abre varios escenarios interesantes en los sistemas de virtualización, presenta ciertas limitaciones, principalmente debido a que la transferencia de un dispositivo físico a una máquina virtual imposibilita que el anfitrión u otra máquina virtual pueda hacer uso del mismo, o expresado de otra forma, inhibe la compartición de recursos hardware. Por este motivo, los fabricantes hicieron un esfuerzo extra en el desarrollo de otra tecnología: *SR-IOV (Single-Root Input/Output Virtualization)*. Esta tecnología se suele conocer como Funciones virtuales o VFs, las cuales no deben ser confundidas con NFV que representan un dispositivo de red completo virtualizado.

Una VF es un dispositivo virtual generado a nivel hardware por el dispositivo físico. De esta forma, el resto de componentes del sistema detectan un dispositivo (especial) independiente, de forma que puede ser asignado a una máquina virtual como cualquier otro dispositivo PCIe. No obstante, una VF es una simplificación

del hardware físico, llamado en este campo como función física (Physical Function (PF)), por lo que carece de cierta funcionalidad. En el caso de las tarjetas de red, las VFs suelen carecer de ciertos contadores básicos o el control sobre el enlace físico. Las VFs no son solo usadas en las máquinas virtuales tradicionales, sino tambien en opciones de virtualización ligeras como los contenedores de Linux. (*Linux Containers*). La tecnología de los contenedores ha sido utilizada por multitud de herramientas, como Docker, que han aparecido en los últimos años.

La diferencia entre la máquina virtual clásica y los contenedores radica en que componentes se virtualizan y cuales permanecen compartidos entre el anfitrión y el invitado. Una Virtual Machine (VM) clásica tiene su propia región de memoria asignada y su propio sistema operativo completo (lo que incluye el Kernel). Por otro lado, un contenedor comparte el Kernel del anfitrión y el uso de memoria está limitado por reglas. Esta compartición de recursos hace que los sistemas basados en contenedores presenten un mayor número de riesgos que una máquina virtual clásica [54]. Por otro lado, su rendimiento es potencialmente superior [55], ya que las llamadas al sistema no requieren un doble o triple cambio de contexto: del usuario al Kernel invitado, del Kernel de invitado al hypervisor y si este no se ejecuta a nivel del Kernel, del hypervisor al Kernel anfitrión.

Dada la importancia del modelo de virtualización, en las siguientes secciones discutiremos acerca de las diferentes alternativas de asociar dispositivos a las máquinas virtuales y su impacto en el rendimiento. En la Figura 2.5 se muestran las diferentes posibilidades.

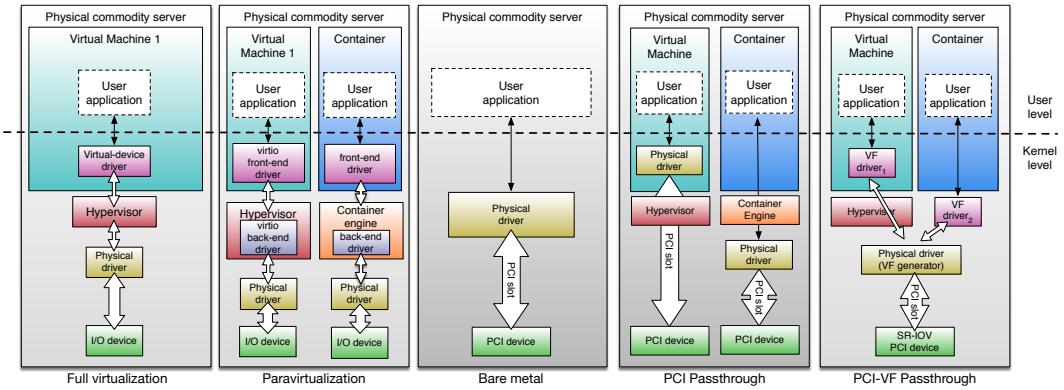


Figura 2.5. Diferentes formas de conectar dispositivo de entrada/salda con una VM o un contenedor Linux.

Virtualización completa

El paradigma de virtualización completa aparece cuando el invitado no es capaz de distinguir si el entorno de ejecución es virtual o no, ya que cada uno de los dispositivos disponibles simula a la perfección un dispositivo físico. Esto incluye, por tanto, que los dispositivos simulen los identificadores, registros y cada detalle y característica del dispositivo a bajo nivel. Este paradigma no es aplicable a los contenedores de Linux, ya que cada proceso es parcialmente consciente de que el entorno de ejecución es virtual, sin contar con que la mayor parte del hardware disponible para estos procesos es físico o paravirtual. Este tipo de virtualización pura es el que en la teoría es más deseable, dado que el invitado es completamente independiente del hypervisor o del hardware subyacente, de modo que una máquina virtual puede ejecutarse uniformemente en un entorno potenciamente heterogéneo.

En el caso práctico, la completa virtualización de dispositivos de red presenta importantes problemas de rendimiento. En este escenario, los paquetes son recibidos por el driver físico de la NIC en el sistema operativo anfitrión y atraviesan la pila de red del Kernel para ser entregados al módulo de red del hypervisor. Una vez en el dominio del hypervisor, el paquete se copia a la VM correspondiente en función de la configuración de red del hypervisor. Nótese que este camino necesita al menos 2 copias adicionales: Una desde el anfitrión al hypervisor y otra desde el hypervisor a la VM, lo que degrada de forma claramente el rendimiento. Por este motivo, la academia y la industria han realizado esfuerzos para optimizar los sistemas y políticas de manejo de paquetes por parte del hypervisor. Utilizando una tarjeta e1000 (una tarjeta Gigabit Ethernet de Intel) “paravirtualizada”, los autores de [56] presentan un switch software (denominado VALE) para incrementar el ancho de banda del sistema. Esta mejora, permite pasar de un máximo inicial de 300 Mbit/s a prácticamente un 1 Gbit/s en el peor escenario (paquetes UDP de 64 Bytes); y entre 2.7 y 4.4 Gbit/s cuando se transmite TCP entre dos máquinas virtuales dentro del mismo servidor físico.

Es importante destacar que cuando se virtualiza completamente un dispositivo o se paravirtualiza, el hypervisor se convierte en un punto central de comunicación y, por tanto, un cuello de botella.

Paravirtualización y VirtIO

La paravirtualización ha aparecido como una alternativa a la aproximación de la virtualización completa. Esta filosofía es similar a la anterior, salvo que la máquina virtual es capaz de identificar los dispositivos virtuales creados por el hypervisor como dispositivos virtuales. En una virtualización tipo KVM, el sistema operativo de la VM debe instanciar un driver compatible, lo que causa una dependencia entre el sistema operativo invitado y el hypervisor No obstante, debido a las decisiones de diseño de estos dispositivos y drivers, las transferencias de datos entre anfitrión e invitado obtienen un mejor rendimiento ya que no es necesario simular detalles de bajo nivel del dispositivo y se pueden llegar incluso a mapear regiones del anfitrión al invitado para evitar copias intermedias. Los drivers VirtIO [57] se han convertido en el estándar de facto para la comunicación entre máquinas virtuales y el hypervisor KVM. El diseño de la API es flexible, permitiendo una comunicación basada en colas de estructuras y funciones *callback*. VirtIO permite crear diferentes tipos de dispositivos Entrada / Salida, como dispositivos de bloques, consolas o dispositivos PCI. En la última década, se ha añadido soporte para dispositivos de red [58]. Este rendimiento también fue comparado en el trabajo citado previamente [56], en donde VirtIO puede llegar a alcanzar 4 Gbit/s utilizando el switch VALE presentado en ese trabajo. Una alternativa posible a VirtIO es el módulo NetVM [59]. Este trabajo obtiene unos resultados muy prometedores (hasta 34 Gbit/s) en ancho de banda entre dos máquinas virtuales instanciadas en el mismo servidor físico. No obstante, no presentan ninguna solución para la captura de tráfico, ni interna ni externa.

Los contenedores Linux, por otro lado, tienen unos dispositivos paravirtualizados mucho más sencillos. Las interfaces de red dentro del contenedor son comúnmente puentes (*bridges*) o dispositivos macvtap, ambos tipos de interfaz son generados por el Kernel y pueden hacer uso de dispositivos físicos. Este tipo de dispositivos son similares a los creados de forma nativa por Open virtual Switch (OvS) [60], los cuales pueden proporcionar anchos de banda interno cercanos al ancho de banda de la memoria [61]. No obstante, aunque ésta implementación es óptima para un uso normal, el OvS requiere un camino de datos no muy eficiente para construir una interfaz de *mirroring*. En términos de eficiencia, la implementación de OvS requiere hasta 6 copias intermedias: La primera, de la aplicación a la interfaz en espacio de Kernel, la segunda, al requerir un tratamiento especial será copiada a un proceso de usuario de OvS. Al decidir el destino del paquete, el

proceso OvS copia al Kernel el paquete por duplicado, para que este sea entregado tanto en la interfaz de destino cómo en la interfaz de monitorización. Finalmente, la quinta y sexta copia del paquete la produce el Kernel al entregar el paquete a la aplicación de usuario final y a la aplicación de monitorización. No obstante, OvS tiene una implementación en DPDK que podría reducir el número de copias a la mitad, es decir a 3: De la aplicación de usuario a una interfaz de DPDK, y por OvS a las dos interfaces especiales de DPDK. Existen similitudes conceptuales en como los drivers VirtIO y los contenedores gestionan los dispositivos E/S. Por ejemplo, en el caso de los dispositivos de almacenamiento, VirtIO mapea un fichero de Linux dentro del espacio de la máquina virtual, mientras que los contenedores funcionan utilizando mapping de ficheros y carpetas para transferir dispositivos del anfitrión a las máquinas virtuales.

PCI passthrough

La mayoría de los fabricantes grandes de microprocesadores, como Intel, AMD y ARM, implementan unidades de control de memoria E/S (*I/O memory management units* (IOMMU)), así como un conjunto de instrucciones para su manejo. El conjunto de instrucciones varía entre fabricantes, en el caso de Intel esta tecnología es llamada VT-d mientras que para AMD es denominada Vi. Estas funciones teóricamente proporcionan la protección y soporte necesarios para mapear de forma segura dispositivos PCI Express (tanto físicos como virtuales) al espacio de memoria de las máquinas virtuales. Esta tecnología es denominada *PCI passthrough*. Al utilizar *PCI passthrough*, al acceso desde la máquina virtual al dispositivo supone un sobrecoste mínimo, ya que todas las capaz de virtualización software intermedias desaparecen. Nótese, que en este escenario el driver que debe ser utilizado en la máquina virtual es el propio driver físico. Como consecuencia, esto permite a una aplicación ejecutada dentro de una máquina virtual beneficiarse de algunas ventajas de ser ejecutada en un equipo dedicado. El uso de *PCI passthrough* ha sido aplicado de forma satisfactoria en entornos de computación de alto rendimiento [62, 63].

El concepto de *passthrough* es más sencillo en el contexto de los contenedores, ya que para otorgar el control completo de un dispositivo a un contenedor basta con mapear los ficheros de control correspondientes al sistema de ficheros del contenedor. Es importante destacar aquí que el driver que el driver utilizado debe ejecutarse nuevamente el anfitrión, ya que el contenedor no puede cargar ningún

módulo del Kernel por razones obvias de seguridad. Por lo tanto, la diferencia de rendimiento entre una aplicación ejecutándose en el anfitrión y otra en el contenedor debe ser mínima. Esto se observará en detalle en la Subsección 2.1.6.

Funciones virtuales PCIe

PCI passthrough sufre una limitación inherente a su diseño: Solo una máquina virtual puede hacer uso de un dispositivo transferido mediante *PCI passthrough*. Por lo tanto, esta tecnología por si sola presenta problemas escalabilidad al incrementar el número de VMs, ya que el número de VMs que puede ejecutarse en un nodo físico depende por tanto den número de interfaces de red físicas disponibles. Esto no es solo un problema de aprovisionamiento de tarjetas de red, sino que al estar los microprocesadores limitados en número de líneas de PCIe, no es posible físicamente conectar más de cierto número de dispositivos PCIe a un cierto procesador. Con el objetivo de promocionar la virtualización de alto rendimiento y la interoperabilidad, el *PCI Special Interest Group* ha desarrollado una serie de estándares denominados *Single-Root I/O Virtualization* (SR-IOV)[3]. En estos estándares se utiliza el término PF para referirse a un dispositivo PCIe conectado físicamente a un slot PCIe. También introducen el concepto de VF cómo un medio para que los dispositivos PCIe puedan ofrecer una interfaz ligera para manejar y compartir parte de los recursos del dispositivo físico. Una VF es un visto por el sistema como otro dispositivo PCIe con ciertas propiedades. Tiene, por tanto, su propia área de memoria asignada para realizar operaciones de E/S, así como un conjunto de registros relacionados con su control y funcionamiento, de modo que el sistema final pueda hacer uso al igual que cualquier otro dispositivo PCIe, aunque típicamente, con menor funcionalidad disponible. Debido a las peculiaridades de las VFs, este tipo de dispositivos generalmente necesitan un driver diferente al utilizado por la PF. Tras la creación de las VFs necesarias, estas pueden ser asignadas al espacio de memoria de una VM utilizando la tecnología anteriormente descrita: PCI passthrough, como si de un dispositivo físico se tratase. La mayor ventaja de la VF frente al uso de PFs es, en función de las capacidades del hardware, una única PF puede producir un elevado número de VFs, incluso cientos en algunos casos. Este comportamiento permite a los gestores de sistemas resolver problemas de escalabilidad asociando nuevas máquinas virtuales a nuevas VFs, sin la necesidad de incrementar o modificar el hardware.

[3] `http://pcisig.com/specifications/iov/`

2.1.3. Monitorización y redes virtuales

La virtualización está cambiando la forma en la que se diseñan y operan las redes. SDN y NFV son ejemplos de estas nuevas tendencias donde el software puede ser finalmente desacoplado del hardware. Un buen ejemplo se muestra en [61], donde los autores exponen diversos escenarios de uso para switches virtuales en los que cuantizan el ancho de banda y la latencia cuando se ven involucrados interfaces tanto físicas como virtuales. En ese trabajo hacen un gran énfasis en OvS [60], una implementación de un switch virtual multicapa que de forma popular se ha convertido en el estándar de facto como switch NFV.

Del mismo modo, la posibilidad de realizar monitorización pasiva dentro de entornos virtuales ha ido creciendo en interés en los últimos años. Uno de los primeros sistemas de monitorización virtualizado fue recogido en la patente [64], en donde una (mal) llamada *virtual* network probe es presentada como sistema de monitorización en redes LTE. Sin embargo, el término usado en esa patente es ligeramente distinto al que presentamos en esta tesis. En la patente consideran *virtual* al sistema solo porque el operador de la red LTE no conoce los detalles de cómo se está realizando la monitorización. ConMon [65] es un sistema automatizado para supervisar el rendimiento de la red sistemas virtualizados con contenedores. En particular, se centra en los efectos de la monitorización de tráfico pasivo y su impacto en el rendimiento y en los recursos del sistema anfitrión. Las sondas virtuales EXFO[4] son soluciones destinadas a desplegar verificadores de NFV sobre una red SDN que ejecutan algunos test para solucionar problemas puntuales. Una idea similar se presenta en [66], en donde los autores diseñan una sonda virtual sobre Open vSwitch para detectar problemas de rendimiento utilizando los nuevos NSH (*network service headers*) con extensiones para la monitorización [67]. No obstante, a pesar de que estos trabajos manejan y presentan sondas virtuales, no presentan ninguna sonda capaz de capturar y almacenar o analizar el tráfico de forma integral. El concepto de sonda virtual aparece de nuevo en [68], y en este caso, los autores introducen algunas de las ideas clave que no habían sido tenidas en cuenta hasta el momento: como se monitoriza el tráfico entre NFVs. Por desgracia, el artículo no explica ningún detalle de implementación, pero aseguran poder realizar inspección profunda de paquetes en enlaces de hasta 10 Gbit/s, con ciertas configuraciones. La arquitectura propuesta muestra

[4] `https://www.exfo.com/en/products/network-performance-monitoring/virtual-probes/`

que este proceso de monitorización se realiza en la capa de virtualización, es decir, en el dominio del hypervisor. La desventaja por tanto de esta aproximación es el propio hypervisor y en como su implementación es capaz de gestionar la copia de paquetes. Volviendo al caso de uso de las redes 5G, es necesario poder alcanzar tasas agregadas de hasta 30 Gbit/s, lo que no parece viable en base a los números aportados en su artículo. Una discusión similar acerca de las limitaciones de la virtualización se expone en [69], en donde DPDK se utiliza para aumentar el ancho de banda y rendimiento de un Open vSwitch. El hardware especializado, como las FPGAs [70], también tiene cabida a la hora de incrementar el rendimiento de las redes virtuales. No obstante, siguen teniendo su desventaja de precio y escalabilidad. Además, en un entorno tipo 5G en donde se busca la homogeneidad a través de la virtualización, el hardware específico podría dar lugar a sistemas heterogéneos virtualizados.

Por otro lado, el uso de máquinas virtuales impone un sobrecoste computacional a las aplicaciones que se ejecutan sobre ellas. Este efecto se aplica en mayor medida a aquellas aplicaciones intensivas en cómputo o con alto número de operaciones en entrada y salida, justo el caso de las NFVs y la VNP. Como ejemplo, una aplicación procesando tráfico de un enlace de 40 GbE puede llegar a tener que procesar hasta 59.52 Millones de paquetes por segundo en el peor escenario. Esto implica una latencia de acceso a memoria extremadamente baja, así como un gran ancho de banda en donde una pequeña pausa (debido, por ejemplo, al hypervisor) puede causar un impacto significativo en el número de paquetes descartados o perdidos. Existen diferentes métodos de virtualización, cada uno con sus ventajas y desventajas y diferentes grados de aislamiento y rendimiento, como se ha discutido en secciones anteriores. Dado que, en nuestro caso de uso particular, en las redes 5G la virtualización no se encuentra restringida ni limitada a un escenario concreto, vamos a explotar diferentes modelos de virtualización y sus hypervisores más populares.

Comúnmente la virtualización se puede dividir en 3 piezas bien distinguidas: El anfitrión (*Host*), el hypervisor y el invitado (*Guest*). El anfitrión es el encargado de comunicarse con el hardware del mundo real, el hypervisor de crear canales de comunicación entre el anfitrión y el invitado creando así el entorno virtual; y finalmente el invitado utiliza el hardware virtual expuesto por el hypervisor. En este modelo clásico, el hypervisor genera un gran aislamiento entre el anfitrión y el invitado, otorgando al invitado una funcionalidad que puede ser completamente independiente del hardware físico. Este método, en cambio, tiene sus sobrecostes,

como la realización de copias dobles. Este concepto de hypervisor ha cambiado desde la aparición de los contenedores de Linux [71], en donde el propio Kernel del anfitrión actúa como un hypervisor ligero y simplificado. Un Contenedor de Linux (comúnmente abreviado LXC) es un entorno cerrado dentro del anfitrión, donde los procesos de invitado y anfitrión comparten el mismo Kernel pero solo pueden interactuar con los sistemas de ficheros y procesos autorizados, pudiendo ser algunos virtuales. Esto implica que puedan presentarse problemas de aislamiento en LXC, ya que un error crítico del Kernel en un sistema invitado potencialmente podría bloquear al anfitrión y al resto de contenedores [54].

Dependiendo de la tecnología de virtualización escogida, deben realizarse ciertos ajustes para optimizar el rendimiento. Dentro de la virtualización clásica, KVM [72] destaca como uno de los más usados y con una ligera ventaja de rendimiento frente a otras soluciones equivalentes [73]. Prueba de ello, es el número de grandes empresas, como Google o Amazon[5], que ofrecen sus servicios en la nube con virtualización en KVM Dentro de la virtualización basada en contenedores, Docker[6] destaca como una de las soluciones más conocidas y flexibles [54]. Por este motivo, en este capítulo nos centraremos en exclusiva en KVM y Docker, como los representantes de sus correspondientes tecnologías.

Las dos optimizaciones principales que deben realizarse en un entorno virtual es la asignación de la CPU y memoria. Cuando un sistema físico tiene más de una CPU, este echo debe ser reflejado correctamente en la configuración de la VM o contenedor. En caso de omitir esta optimización, la VM presentará un uso de recursos ineficiente, ya que existirá comunicación entre las CPUs, que causa incrementos en la latencia de acceso a memoria y por ende degradación del rendimiento. Existe otra optimización que debe ser tenida en cuenta y que afecta solo a los sistemas de virtualización tradicionales, ya que un elemento crítico está siendo virtualizado: La memoria. Ya que la VM es un proceso al igual que cualquier otro, su memoria se encuentra segmentada en páginas. No obstante, el sistema operativo invitado a su vez segmentará la memoria de los procesos en páginas, creando un doble nivel de paginación. Para mitigar el impacto de este efecto se debe recurrir al uso de *hugepages* por parte del hypervisor [74]. Finalmente, y si el hardware del anfitrión lo soporta, deben ser habilitadas las instrucciones de virtualización.

[5] `https://aws.amazon.com/ec2/faqs/?nc1=h_ls`
[6] `https://www.docker.com/`

2.1.4. Diseño de un driver 40GbE: HPCAP40 y HPCAP40vf

Como ya ha quedado claro en secciones anteriores, DPDK se ha establecido como uno de los motores de captura y gestión de tráfico predominantes en el mercado. No obstante, portar las aplicaciones sobre otros sistemas de captura anteriores requiere de una evaluación exhaustiva. Uno de los elementos a verificar es su propia arquitectura y diseño, que si bien se encontraba bien evaluada para tarjetas de 10 Gbit/s [7, 11, 12], no había sido evaluada con tecnología más novedosa. Por este motivo se decidió evolucionar HPCAP y portarlo a tarjetas de 40 Gbit Ethernet de Intel, en particular al modelo Intel XL700, ya que nuestro objetivo es la monitorización en redes 5G, que son de al menos 40 Gbit/s. Y lo hemos llamado HPCAP40.

Uno de los factores de diferenciación entre DPDK y la filosofía aplicada en HPCAP es el uso del paradigma *onecopy*. Aunque el paradigma *zerocopy* suele ser más eficiente en términos generales, es más sencillo y menos peligroso mapear regiones intermedias del Kernel al usuario. Cada gestor de paquetes a alta velocidad fue diseñado con un objetivo en mente: routers y switches software, análisis en GPUs, etc. El caso de uso de HPCAP fue el almacenamiento en disco de todos y cada uno de los paquetes, para, posteriormente, realizar un análisis que potencialmente no podría ser ejecutado en tiempo de captura. No obstante, no existe ningún dispositivo con la velocidad necesaria para almacenar a una tasa cercana a los 10 Gbit/s y menos aún durante días. Por eso es necesario recurrir a agrupaciones de discos Redundant Array of Independent Disks (RAID) (ya sea de discos mecánicos, SSD o NVMe). Como cualquier otro dispositivo de bloques, para maximizar utilizar todo el ancho de banda disponible es necesario hacer escrituras y lecturas del tamaño del bloque del dispositivo. En el caso de los RAID, este tamaño de bloque debe ser un múltiplo del tamaño del bloque de cada uno de los discos que lo forman multiplicado por el número de discos que forman el RAID. Si bien tenemos en cuenta que tanto los discos como potencialmente tarjetas RAID hardware implementan colas internas, es posible y necesario paralelizar escrituras de bloques, lo que significa que cada escritura a nivel de aplicación debe tener N veces el tamaño del bloque del RAID, en donde N es una característica propia de los discos utilizados y documentada por el fabricante. Por otro lado, las tarjetas de red utilizan internamente estructuras de memoria para almacenar los paquetes de tamaño fijo y típicamente de 2048 Bytes. Este número no es arbitrario, ya que al ser exactamente 32 líneas de caché y la mitad de una página de memoria

típica de 4098 Bytes (Recordemos que existen *Hugepages* de 2 MiBs y de 1 GiB) se puede obtener un término medio entre memoria consumida y rendimiento de acceso a memoria. Ya que cada paquete de red tiene un tamaño diferente no es posible disponer de una región contigua de memoria en la que la tarjeta pueda escribir los paquetes capturados y que no existan espacios en blanco arbitrarios entre ellos.

Realizar una copia de los paquetes a un buffer intermedio es una pieza clave para poder almacenar los paquetes en un RAID de discos a tasa de línea. Si bien esta copia puede ser realizada nivel de usuario, mantener a una aplicación de usuario un conjunto arbitrario de regiones de memoria procedentes del Kernel es un sobrecoste de recursos que puede ser omitido (haciendo la copia intermedia en espacio del Kernel) si el proceso de monitorización siempre debe realizar este tipo de copia previa. Es por este motivo por el cual se consideró el método *onecopy* como el óptimo en este escenario. Desde el punto de vista del mantenimiento de los ficheros almacenados, decidimos optar por un tamaño de fichero capturado de 2 GiB y nombrar a los ficheros utilizando un *timestamp* (marca temporal) para facilitar búsquedas de ciertos segmentos de tráfico, así como para facilitar la rotación y borrado de viejos ficheros. Para mantener los ficheros de tamaño constante, así como las operaciones de escritura, si un paquete es demasiado grande para entrar al final del fichero que va a ser escrito a continuación, se agrega un pequeño relleno hasta alcanzar los 2 GiB. La versión HPCAP original [11] además de almacenar los paquetes capturados en ficheros, soporta conectar varias aplicaciones a una misma interfaz y realizar algunos análisis sobre los paquetes, cómo la reconstrucción de flujos. La versión que se realizó de 40 Gbit mantiene esta misma API a la vez que esta funcionalidad.

Acceso a la tarjeta de red

Tal y como se ha discutido en secciones anteriores, la mejor opción de paralelizar la captura de tráfico en la recepción es el uso de múltiples colas RSS [39]. El funcionamiento de las colas RSS se fundamenta en el uso de hashes sobre la quíntupla, de forma que se puede asegurar que un mismo hash (o flujo) siempre va a ser asociado a la misma cola de recepción RSS, no obstante, no se puede asegurar de ninguna forma que el tráfico entre dichas colas vaya a encontrarse equilibrado, pues existe una relación con el tráfico subyacente que puede ser eliminado. Para poder hacer un buen análisis sobre los paquetes capturados, es necesario conocer

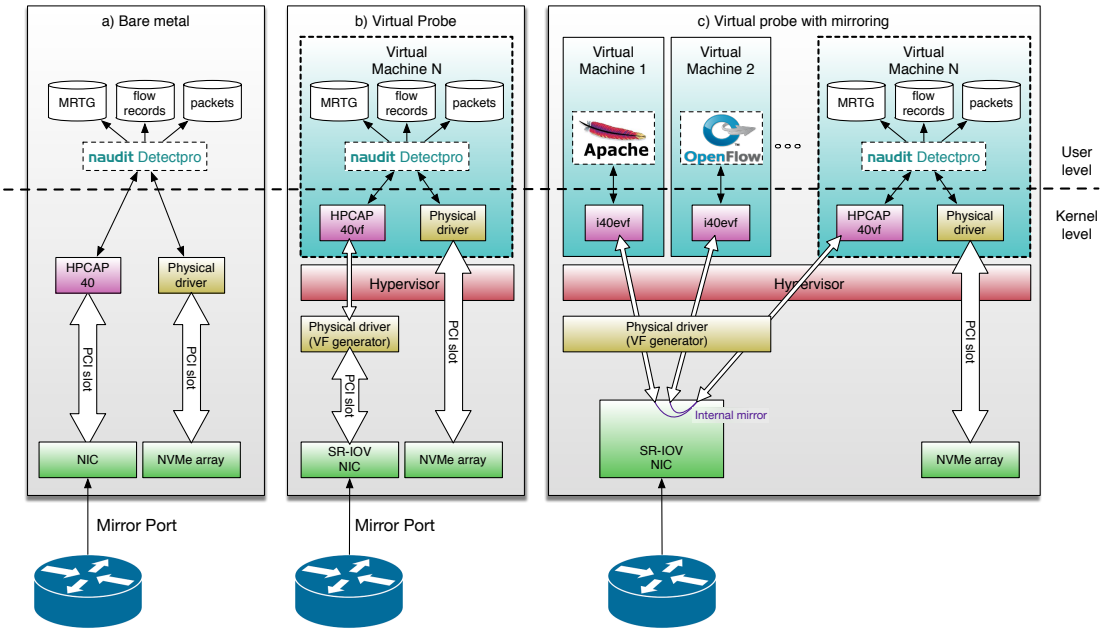


Figura 2.6. Ejemplo de despliegue de sondas. a) Bare metal, b) Virtual probe y c) Virtual probe mediante mirroing.

con precisión su momento de recepción. Típicamente esto solo es necesario dentro de un flujo, no obstante, la definición de flujo supone un problema, ya que, si consideramos quíntupla como la definición estricta de flujo, una conexión TCP entre dos equipos puede ser entendida como dos flujos diferentes, uno por cada sentido. El uso de colas RSS supone un reto en el análisis a nivel de conexión, ya que cada núcleo de CPU es consistente a la hora de marcar la recepción –estrictamente creciente–, pero no tiene por qué serlo con respecto a otros núcleos de CPU, de forma que en el proceso de monitorización puede producirse desordenamiento de paquetes, lo que es un problema a la hora de analizar la conexión a nivel TCP. Por ese motivo se comenzó a explorar alternativas de paralelización a bajo nivel que no necesitasen colas RSS. La primera y más sencilla aproximación es utilizar el método *first-come-first-served* (FCFS), es decir, cada thread bloquea la cola de recepción y lee el primer paquete que esté disponible. Sin embargo, esto causa dos problema: una sincronización continua entre hilos que degradará el rendimiento y el mismo desorden producido en las copias de paquetes. Una posible solución es asignar regiones de la cola (ya que todos los descriptores de paquete comparten el mismo tamaño) y asignarla a un determinado hilo tal y como se muestra en la Figura 2.6. De esta forma, no es necesario utilizar ningún sistema de sincroniza-

ción entre hilos de captura, ya que cada hilo debe hacerse responsable únicamente de su región, manteniendo un sistema favorable a las caches, con una latencia y potencia de procesamiento predecibles. El único desordenamiento posible de paquetes se puede producir cuando dos hilos se encuentran copiando al buffer intermedio un paquete en el borde del sector asignado al otro proceso. Por tanto, aunque el desorden sigue presente, se encuentra en un factor inferior comparado con el uso de colas RSS.

No obstante, en este diseño sigue siendo necesario introducir un punto de sincronización. Los sectores de la cola de paquetes utilizados por cada hilo deben ser devueltos a la interfaz (para que introduzca nuevo paquete) en el mismo orden en el que fueron escritos, de esto modo, ningún hilo se adelantará a otro y se asegurará la entrega ordenada. Con el objetivo de realizar pruebas comparativas se implementó el uso de colas RSS y se asignó igual que otros sistemas de captura una cola única por hilo de recepción. En este caso, no es necesaria hacer la sincronización anterior, pero a cambio se van a producir todos los eventos no deseables discutidos anteriormente: Desordenamiento, problemas de marcado temporal, perdidas de tráfico puntuales debidas a tráfico no distribuido perfectamente, etc.

Escritura al buffer intermedio

Uno de los pasos críticos en el funcionamiento de HPCAP40 es la propia coordinación de los hilos de captura en la escritura en el buffer intermedio. Dado que todos los hilos deben escribir en el mismo buffer es necesario utilizar un método con el mínimo coste en sincronización posible. Un modelo posible de implementación es ver este buffer como un caso de cola de múltiples lectores con múltiples escritores. Sin embargo, gracias a ciertos aspectos del driver es posible realizar algunos atajos y optimizaciones para mejorar el rendimiento. El método básico de sincronización consiste en utilizar una variable atómica que apunte al punto donde se escribirá el próximo paquete. Esta variable es incrementada por cada hilo cada vez que escribe un paquete, permitiendo que todos los hilos puedan escribir de inmediatamente de forma efectiva. Dado que el HPCAP controla el tamaño del buffer intermedio, no es necesario utilizar un sistema de sincronización especial para evitar que las escrituras salgan fuera de la región del buffer. Gracias a las *hugepages* es posible reservar memoria físicamente contigua de 2^{32} bytes (4 GiB), de forma que direccionando esta memoria con una variable de 32 bits. De esta

forma, el desbordamiento del entero causa de forma automática que el contador vuelva al inicio del buffer evitando realizar operaciones modulares (divisiones).

La misma idea puede utilizarse para controlar los offsets dentro de un fichero para el control del padding. El problema a resolver es debido a que el espacio de memoria del buffer intermedio debe ser utilizado para escribir 2 ficheros de 2 GiB cada uno. Una posible solución es cambiar el método de sincronización de los hilos para que tengan estos efectos en cuenta y añadan cuando sea preciso espacios en blanco al final del fichero si el paquete en cuestión superase el límite de los 2 GiB del fichero actual. Otra posible aproximación es reservar siempre un espacio al comienzo y al final del fichero, marcando desde un inicio que no contiene un paquete válido. La decisión de que hilo debe encargarse de la reserva del espacio vacío es delicada, ya que otro hilo podría reservar poco espacio, impidiendo la colocación de la cabecera PCAP al inicio o podría haber reservado demasiado el hilo anterior, impidiendo la construcción de un paquete vacío válido al final del fichero.

API para clientes

Siguiendo el modelo de cola con múltiples productores y consumidores, debería existir una sincronización entre hilos de lectura y escritura. La arquitectura del driver permite realizarlo de forma simple. Para evitar sobre-escrituras, el driver posee un offset global dentro del buffer compartido, marcando el último byte que ha sido leído por el cliente más lento. Esto permite el cálculo de le espacio disponible e en el buffer, de modo que cada hilo puede escribir hasta un máximo de $\lfloor e/n \rfloor$ bytes sin necesidad de volver a consultar la variable global, siendo n el número de hilos de captura. Esto permite evitar lecturas innecesarias de la variable atómica cada vez que se recibe un paquete, ya que, aunque el acceso a estas variables es relativamente ligero, sigue teniendo un impacto al hablar de hasta decenas de millones lecturas por segundo. De forma similar, la aplicación a nivel de usuario utiliza las variables atómicas de los diferentes hilos de captura para no acceder nunca a una región de memoria que aún no se haya escrito.

Almacenamiento en disco

Cada cliente, tal y como se ha explicado anteriormente, tiene un puntero al primer byte que puede leer, el número de bytes disponibles y notificará periódicamente al driver de cuantos bytes ha leído. Existe por tanto completa libertad para

el cliente de gestionar para leer y procesar los paquetes. La aproximación de lectura clásica viene heredada del driver HPCAP original de 10 Gbit/s. Una aplicación copia los datos del buffer intermedio directamente a ficheros en bloques afines al RAID de, por ejemplo, 4 MiB, y cierra el fichero tras almacenar 2 GiB. Esta aproximación es efectiva siempre y cuando el sistema de ficheros y el RAID sean capaces de almacenar a la tasa de captura en el enlace de red. No obstante, almacenar datos a 40 Gbit/s supone un reto difícil de superar por los clásicos discos mecánicos y los enlaces tipo SATA. En este contexto es necesario recurrir a un RAID de discos Non-Volatile Memory Express (NVMe). Un disco NVMe es, de forma resumida, un disco SSD de alta velocidad conectado por PCIe. Este tipo de interconectividad lo convierte en un disco peculiar, ya que internamente utiliza colas de transmisión de bloques muy similares a paquetes de red, pero de un tamaño bastante mayor y siempre múltiplo del tamaño de bloque utilizado internamente. Para explotar estos discos al máximo, es necesario utilizar técnicas similares a las tarjetas de red, de forma que se excluya la pila de llamadas del sistema y todo quede en una capa a nivel de usuario.

Para resolver este problema, Intel desarrolló Storage Performance Development Kit (SPDK)[7], que al igual que DPDK está orientado a proponer drivers a nivel de usuario evitando al máximo el uso de cambios de contexto y funcionalidad genérica innecesaria para una aplicación de almacenamiento concreta. Usar SPDK con NVMes supone un reto, pues no solo es necesario manejar colas de lectura y escritura de bloques de datos, sino que la carencia de un sistema de ficheros implica desarrollar –al menos– un índice en el que localizar los diferentes ficheros a lo largo del RAID. Al iniciar el desarrollo de HPCAP40 [75], hicimos uso de SPDK obteniendo unos resultados razonables pero su uso presentó problemas de estandarización con otras herramientas, pues todas y cada una de ellas debía ser portadas a SPDK para poder acceder a las trazas de red almacenadas por HPCAP40. La solución a ese problema, por suerte, fue sencilla y consistió en utilizar discos NVMe más rápidos y una versión del Kernel con una implementación de los drivers NVMe y MDADM (RAID software) más modernos y optimizados que nos permitiesen alcanzar los 40 Gbit/s en escritura con un RAID y sistema de ficheros estándar.

[7] `http://www.spdk.io/`

Filtrado de paquetes

A pesar de que la arquitectura previamente descrita permite la captura a tasas muy elevadas de una forma escalable, se ha diseñado para soportar el peor caso posible para 40 Gbit/s, también es interesante tener en cuenta que no siempre va a ser posible disponer de todo el hardware necesario para capturar y almacenar a tasa de linea. Por este motivo se ha añadido un sistema de filtrado sencillo y rápido al driver. Este, permite definir hasta 128 reglas en donde cada una de ellas se define como un listado de bytes que debe ser comparado en una cierta posición del paquete. Estas reglas pueden hacer al driver descartar y no copiar al buffer intermedio el paquete en función de la presencia de dichos bytes. Esto es útil para, por ejemplo, excluir todo aquel paquete UDP que tenga como puerto origen o destino 80 u 8080, o permitir siempre todo aquel paquete TCP independientemente de su puerto. Otra posibilidad para reducir el ancho de banda necesario para el almacenamiento es el truncado de paquetes. De esta forma, el driver solo copiaría hasta un máximo de bytes definido por el usuario al buffer intermedio.

HPCAP en entornos virtualizados

La mayoría de dispositivos de red actuales soportan SR-IOV, donde cada interfaz virtual tiene asociada su propia cola virtual, denominado por los fabricantes como *Virtual Machine Device Queues* (VMDq). El número de funciones virtuales generadas por un dispositivo físico está limitado por el hardware, siendo 128 el número de funciones virtuales soportadas por la Intel XL710. Por tanto, el driver de captura debe ser capaz de gestionar la VF, un aspecto importante, ya la mayoría de sistemas de captura para 10 Gbit/s explicados con anterioridad no tienen capacidad de funcionar en entornos virtualizados a 10 Gbit/s. DPDK, en cambio, si tiene soporte nativo para utilizar VFs, al igual que Intel proporciona un driver nativo para utilizar las PFs (i40e) y otro para las VFs (i40evf).

Para esta interfaz en concreto, se ha construido una versión adaptada del driver de captura HPCAP40, que denominaremos HPCAP40vf, siguiendo los conceptos aplicados en las versiones física [11] y virtual [63] del driver HPCAP. Dado que en el momento de escritura de esta tesis no existe ningún otro driver para capturar tráfico desde una tarjeta Intel XL710 a excepción de los drivers originales, DPDK y la nueva versión de HPCAP, se ha realizado un estudio de rendimiento de los tres, tanto en entornos físicos como virtuales.

El driver de la PF es el encargado de configurar el hardware de la tarjeta para que genere las VFs necesarias y realizar una básica configuración de las mismas. El proceso de construcción de VFs varía entre versiones de drivers y entre modelos y fabricantes de tarjetas. La tarjeta Intel XL710 implementa un pequeño switch de nivel 2 interno, esto permite que el tráfico sea enrutado internamente entre diferentes colas virtuales y la interfaz física sin ninguna intervención por parte del driver de la PF o un proceso de usuario y sin ninguna pérdida de paquetes. No obstante, antes de entrar en la virtualización de redes de 40 Gbit/s, realizamos un estudio similar sobre redes virtuales de 10 Gbit/s [63]. A diferencia de las tarjetas Intel de 40 Gbit/s, las tarjetas de 10 Gbit/s carecen de un switch interno por lo que el microcontrolador debe encargarse del enrutado interno de forma procedural. Debido a sus limitaciones de este proceso, el chipsed sufría de elevadas pérdidas de paquetes. Ya que el fabricante conocía estas limitaciones, permitieron que DPDK –ya que estaba directamente dirigido por DPDK– pudiese no solo controlar la generación de estas interfaces virtuales, sino que podía tomar el control del proceso de switching interno de la tarjeta, obteniendo por tanto un rendimiento superior a la gestión realizada por el propio chipset. La aplicación encargada de hacer este tipo de switching viene de serie con DPDK y se denomina *testpmd*. Para evaluar el impacto que tenía el sistema de reencaminación de paquetes se hizo una serie de experimentos para comparar el rendimiento del rencaminamiento utilizando DPDK frente al propio del chipset y gestionado por el driver *vanilla* de Intel. En la tabla 2.1 se muestran los resultados obtenidos y comparados a su vez frente al método de captura de tráfico posterior: tcpdump con *ixgbevf* –el driver *vanilla virtual*–, HPCAPvf [63] y DPDK. Los resultados obtenidos en el estudio dejan claro que solo DPDK es capaz de gestionar el reenvío de paquetes de forma eficiente entre VFs y el único capaz de asegurar la captura del 100% del tráfico en un entorno de 10 Gbit/s. No obstante, el precio de usar DPDK como método de enrutado requiere de un elevado coste computacional: al menos es necesario destinar un núcleo CPU completo por cada dos VFs en el sistema para asegurar la tasa.

Por motivos de seguridad, las VFs por defecto tienen unas políticas de distribución de tráfico basadas en direcciones MAC (del inglés Media Access Control (MAC)) e IP, las cuales a su vez se encuentran asociadas a otra VF o en su defecto a la PF. Por ello, cada VF solo puede recibir el tráfico que ha sido destinado a su correspondiente VM. Esta limitación es necesaria en la mayoría de los casos, pero es un problema en un escenario de monitorización, en donde es necesario que una

Motor de captura	% de paquetes procesados	
	Chipset	DPDK
ixgbevf	>0.1	1
HPCAPvf	36.2	82.7
DPDK	37.6	100

Tabla 2.1: Porcentaje de paquetes capturados en función del método de encaminamiento de la VF. El tráfico utilizado estaba formado por paquetes de tamaño mínimo (60 Bytes).

de las VMs pueda recibir el tráfico del resto de VFs. Utilizando un conjunto de registros del chipset de la tarjeta (o con ayuda de la interfaz debugfs del driver i40), un gestor puede introducir una regla de *mirroring* al switch interno para replicar el tráfico de ciertas (o todas) las VFs a una determinada VF de monitorización. Nótese, que a diferencia de las tarjetas de 10 Gbit/s, la Intel XL710 puede hacer la replicación a nivel hardware, sufriendo un deterioro del rendimiento mucho menor. En cualquier caso, el rendimiento global del sistema puede verse afectado por las capacidades del switch interno y por el ancho de banda del bus PCIe. Es un efecto a tener en cuenta, ya que al ser cada VF un dispositivo PCIe independiente, cada paquete monitorizado debe ser transferido dos veces, lo que supone una utilización de ancho de banda superior al uso normal.

2.1.5. Arquitectura de una sonda VNP

Cómo se ha explicado con anterioridad, la monitorización pasiva tradicional consiste en conectar sondas a puertos espejo de determinados switches o routers y recibir todo aquel tráfico que cumpla algún tipo de característica (VLAN, puerto de entrada al switch, etc.) (ver Figura 2.6a). Aunque esta aproximación debería proporcionar el mejor rendimiento debido al uso de hardware dedicado, existen diversas situaciones en las que no puede utilizarse. Por ejemplo, al existir varios puntos de monitorización, desplegar multitud de sondas físicas puede convertirse en una tarea compleja y económicamente cara. En estos casos, una alternativa más asequible es utilizar el hardware disponible en donde ejecutar las tareas de monitorización. En este contexto se propone el concepto de VNP, el cual puede ser desplegado en cualquier nodo de cómputo de un centro de datos o en cualquier nodo de infraestructura de red con capacidad para ejecutar máquinas virtuales. En esta sección se describe el proceso de monitorización de una red parcial o total-

mente virtualizada, así como posibles configuraciones de despliegue de las sondas en función de los requisitos de la monitorización.

Despliegue de una VNP

El concepto de VNP propuesto en este capítulo consiste en dos componentes, el propio driver de captura y la herramienta de análisis. En un entorno de producción, la herramienta clásicamente utilizada podría ser algo similar al DetectPro [76] de Naudit; una aplicación que genera registros de flujos en tiempo real y los almacena junto con los paquetes en crudo para un posterior análisis. Sin embargo, cualquier tipo de software puede ser utilizado para analizar los datos, como una herramienta específica de análisis estadístico o una herramienta que simplemente vuelque los paquetes en disco. De igual modo el driver de captura puede variar, al igual que se puede utilizar las diferentes versiones de HPCAP40 y HPCAP40vf desarrolladas en este capítulo se pueden utilizar otras herramientas del estado del arte como DPDK.

Ambos componentes son empaquetados en una máquina virtual o contenedor que podrán ser utilizados para desplegarse directamente en nodos de cómputo. Estos nodos solo necesitan asignar una VF a la VNP, la cual comenzará a monitorizar todo el tráfico que es entregado a la PF. Este escenario se muestra en la Figura 2.6b. En el caso de que no exista una red física clásica que monitorizar, la configuración a realizar será similar a la mostrada en la Figura 2.6c. En este caso, la tarjeta de red hace uso de sus funciones de mirroring avanzadas para retransmitir tanto el tráfico de todas las demás VFs a la VF de monitorización como el tráfico sin destino proveniente de la red física.

Los dispositivos de almacenamiento son, en todos los casos, un conjunto de discos NVMe en RAID 0 capaces de proporcionar tanto la capacidad como la velocidad necesaria. Este espacio puede ser enlazado con la VNP utilizando una de las técnicas descritas en la sección anterior: una configuración puramente física, utilizando PCIe passthrough o mediante paravirtualización con VirtIO. La decisión final dependerá de los anchos de banda alcanzados y las capacidades del nodo de computación. Tal y como se describe en secciones futuras, el escenario con una configuración completamente física fue el que obtuvo los resultados óptimos. Cómo se ha explicado anteriormente, en los test realizados solo se ha considerado ejecutar la VNP sobre una plataforma KVM y Docker. No obstante, el concepto propuesto de VNP no está restringido a unas plataformas específicas de virtuali-

zación, ya que solo es necesaria la presencia de funcionalidad estándar, como la capacidad de realizar PCIe passthrough de ciertos dispositivos o compartir parte del sistema de ficheros entre anfitrión y sonda virtual.

2.1.6. Pruebas de rendimiento y resultados

Una vez que la arquitectura y la tecnología subyacente de la VNP ha sido discutida, es posible presentar las pruebas y resultados obtenidos, y mostrar que el concepto de VNP es posible en redes 40 Gbit/s utilizando tarjetas de red comunes en las futuras redes 5G.

Todas las pruebas realizadas en este capítulo se han realizado en el mismo entorno, consistiendo este en dos máquinas físicas independientes conectadas por un cable de 40 GbE de cobre y cuyas especificaciones se muestran en la Tabla 2.2. Ambas máquinas corren el mismo sistema operativo, un Gentoo mínimo con un Kernel de Linux 4.14.7, el cual tiene desactivados los parches de seguridad de Meltdown y Spectre por cuestiones de rendimiento. Todas las VMs utilizadas corren el mismo sistema, un CentOS 7.4 con un Linux 3.10.0-693.

	Generador de tráfico	**VM Anfitrión**
CPU	2 × Intel Xeon E5-2630v4	2 × Intel Xeon Gold 6126
Reloj	2.20GHz	2.60 GHz
Núcleos	20	24
Memoria	126 GB	188 GB
NIC	Intel XL710	Intel XL710

Tabla 2.2: Especificaciones de los servidores utilizados. HyperThreading está desactivado.

Motor de captura	**Porcentaje de paquetes capturados**			
	FÍSICO		PASSTHROUGH	
	64 B	*CAIDA [77]*	*64 B*	*CAIDA [77]*
i40e	1.6	11.59	0.2	9.44
DPDK	100	100	100	100
HPCAP40	75.1	100	51.5	100

Tabla 2.3: Porcentaje de paquetes capturados con diferentes patrones de tráfico en un enlace 40 Gbit/s completamente saturado utilizando soluciones dedicadas y soluciones con virtualización y PCIe passthrough.

A modo de línea base, se realizó una prueba de rendimiento en las que se medió el porcentaje de captura utilizando DPDK y el driver HPCAP40 sobre una interfaz física, tanto en un entorno dedicado como en uno virtualizado mediante PCIe passthrough. en el primer caso, los drivers se ejecutan sobre el anfitrión, mientras que en el segundo dentro de una máquina virtual. Es interesante poner a prueba el driver original *i40e*, ya que marca el rendimiento mínimo que cualquier aplicación podría alcanzar. Como programa de captura se utilizó *tcpdump* ya que presenta en 10 Gbit/s un rendimiento aceptable en algunos casos y es una herramienta de captura estándar. Como se observa en la Tabla 2.3, el driver *vanilla* no alcanza una tasa superior al 12% de paquetes capturados en ningún caso. DPDK, en cambio, captura el 100% de los paquetes en todos los casos, mientras que el driver de HPCAP40 solo alcanza a capturar el 100% con la traza de CAIDA. Las pérdidas del driver de HPCAP40 con los paquetes de tamaño mínimo se debe al comportamiento de DPDK y HPCAP40 al no tener ningún cliente almacenando. En el caso de DPDK, los paquetes capturados son inmediatamente descartados y su memoria liberada, mientras que el comportamiento de HPCAP40 consiste en copiar igualmente dicho paquete al buffer intermedio, añadiendo un sobrecoste.

Además del driver vanilla, es necesario verificar que el sistema de almacenamiento (en este caso RAIDs NVMe) pueden soportar el ancho de banda tanto en entornos virtualizados como dedicados. Para ello, se realizaron pruebas en diferentes entornos: (I) prueba en un entorno dedicado en donde el RAID construido por la herramienta *mdadm* y montado directamente sobre sobre la máquina anfitrión, (II) el mismo RAID software montado en una máquina virtual con los dispositivos NVMe individualmente transferidos mediante passhtrough, y (III) el RAID se monta en el anfitrión y se transfiere a la máquina virtual utilizando paravirtualización (VirtIO). Sobre cada uno de estos escenarios, se copian 3000 bloques de 5635 KB cada uno (Este es el tamaño de bloque óptimo de acuerdo a las especificaciones de los discos NVMe para obtener su máximo rendimiento de escritura) y anotando el ancho de banda obtenido. La prueba se ejecuta 10 veces.

El sistema utilizado en esta prueba fue el “VM Anfitrión” (ver Tabla 2.2) que posteriormente se utilizará como equipo de captura. En este equipo están disponibles 11 discos NVMe Intel DC P3700. La elección de este modelo en concreto de dispositivos fue debida a que en el momento de la compra del equipamiento era el medio de almacenamiento con mejor velocidad de lectura y escritura a un precio razonable. Remplazar los dispositivos NVMe por discos mecánicos tradicionales es posible, pero sería necesario recurrir a RAIDs con un número elevado de discos

que posiblemente superasen los 40. Esto convierte a este tipo de almacenamiento en una solución poco practica y comparativamente económicamente más cara.

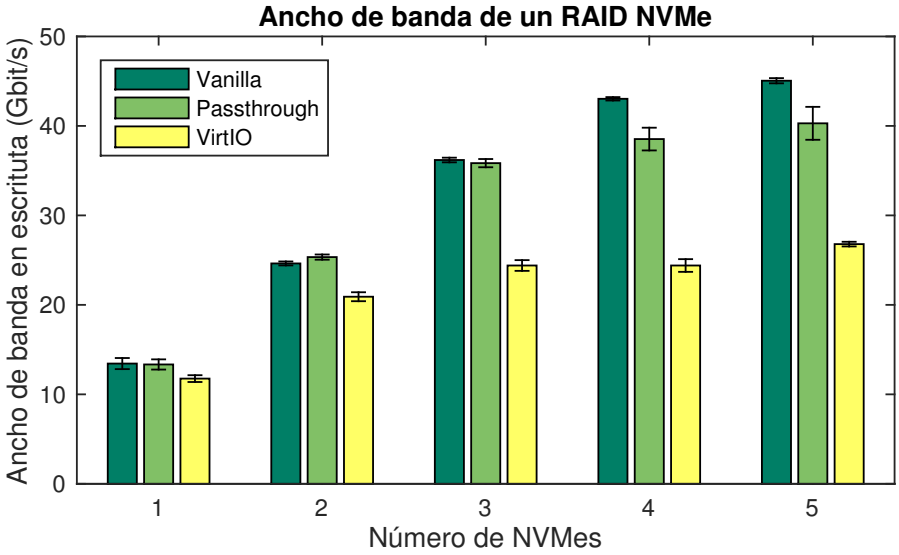


Figura 2.7. Ancho de banda utilizando un raid software *MDADM* en función del número de NVMes utilizados.

Los resultados obtenidos se muestran en la Figura 2.7, donde solución dedicada (*vanilla*) destaca por superar ampliamente los 40 Gbit/s con 5 dispositivos. Con 5 dispositivos al método de PCIe passthrough le cuesta alcanzar la tasa de 40 Gbit/s sostenidos y tiene una gran variabilidad, mientras que la paravirtualización con VirtIO deja de crecer linealmente en 3 discos con una tasa de escritura de aproximadamente 25 Gbit/s.

Tras confirmar que el equipamiento de "VM Anfitrión" ejecutando los procesos de captura y almacenado obtiene unos resultados razonables, es posible avanzar al siguiente paso y probar tanto HPCAP40vf como DPDK en dos escenarios, cada uno de ellos relacionado con un punto crítico en la captura de tráfico en VFs. Lo primero a evaluar es que es posible recibir paquetes de red en una sonda virtual a la misma tasa del enlace físico y probar que el driver de captura virtual funciona bien y no se encuentra afectado de forma notable por las restricciones del entorno virtual en el que se ejecuta. Adicionalmente, es interesante medir el efecto de una captura entre diferentes máquinas virtuales. Lo segundo que es necesario evaluar son los sistemas de almacenamiento que, en paralelo con la captura, deben ser capaces de funcionar sin representar un cuello de botella.

Captura de tráfico en una VF

Para poder medir la capacidad del sistema recibiendo tráfico desde una VF y los efectos que esta interfaz pueda causar en la medida, hemos establecido nuestro banco de pruebas de la siguiente forma. Una de las máquinas envía tráfico sintético de tamaño fijo saturando el enlace de 40 GbE hacia el equipo de captura. Este equipo ejecuta 3 máquinas virtuales en KVM, cada una de ellas asignada a su propia VF. La primera VM captura el tráfico procedente del driver y lo descarta sin almacenar, mientras que las otras dos máquinas virtuales ejecutan un test de ancho de banda con iperf3[8]. La primea VF es configurada con modo promiscuo y con las reglas de espejo necesarias para copiar el tráfico de las otras VFs, de modo que por esta interfaz virtual circule tanto el tráfico externo como el interno. A modo de referencia, iperf3 reporta un ancho de banda de 25 Gbit/s entre ambas máquinas virtuales cuando no se encuentra ninguna sobrecarga externa ni proceso de captura.

Cada prueba realizada con tráfico sintético envía paquetes de tamaño constante. La Tabla 2.4 muestra los resultados de estos test para los tamaños de 64, 128, 256, 1024 y 1518 Bytes a nivel Ethernet incluyendo CRC. Para paquetes de tamaño 64 Bytes, DPDK captura el 66.3 % de los paquetes mientras que HPCAP40vf un 57.5 %. No obstante, el número de paquetes por segundo es tan elevado con estos tamaños que el switch interno de la tarjeta no es capaz de lidiar en paralelo con el test de iperf, por lo que este queda limitado a tan solo 66.7 Kbps, pudiendo decirse, por tanto, que este es el ancho de banda disponible entre dos máquinas virtuales cuando una tercera se encuentra realizando monitorización en el peor caso posible. Los resultados mejoran al aumentar ligeramente el tamaño del paquete ya que con 128-Bytes, DPDK es capaz de capturar el 100 % del tráfico, HPCAP40vf se queda en apenas un 61.9 % mientras que el ancho de banda entre máquinas virtuales se incrementa a 2.2 Gbit/s. La diferencia entre los drivers de captura era esperable, ya que HPCAP40vf está optimizado para el almacenamiento lo que implica una copia extra que no se realiza con DPDK, incluso si la prueba consiste en capturar y descartar paquetes.

Con paquetes de tamaño 256-Bytes, ambos DPDK y HPCAP40vf, son capaces de capturar el 100 % del tráfico sin pérdidas, mientras que el ancho de banda entre máquinas virtuales se incrementa a 3.5 Gbit/s. Como se observa, incrementar más el tamaño del paquete no mejora de manera notable el ancho de banda medido por

[8] `https://iperf.fr/`

Tamaño del paquete	Tasa de TX	DPDK	HPCAP40vf	iperf
64 bytes	59.5 Mpps	66.3%	57.5%	66.7 Kbit/s
128 bytes	33.8 Mpps	100.0%	61.9%	2.2 Gbit/s
256 bytes	17.9 Mpps	100.0%	100.0%	3.5 Gbit/s
1024 bytes	4.7 Mpps	100.0%	100.0%	3.8 Gbit/s
1518 bytes	3.2 Mpps	100.0%	100.0%	3.6 Gbit/s

Tabla 2.4: Rendimiento de captura en una VF en la que, se captura tráfico sintético de diferente tamaño procedente del enlace físico en paralelo con la ejecución de dos máquinas virtuales comunicándose entre sí a través de un iperf en el mismo equipo.

iperf3. Este efecto no se debe a una limitación de la potencia del switch de la tarjeta sino a una limitación del bus PCIe y como la tarjeta es capaz de gestionarlo, dando un límite aparente de unos ~44 Gbit/s, encontrándose este por debajo del teórico de los 63 Gbit/s del que dispone el bus PCIe 3.0 de 8 líneas al que se encuentra conectada la tarjeta.

Estas mismas pruebas se ejecutaron en un contenedor Docker. No obstante, es necesario recordar que una VNP funcionando sobre contenedores tiene al driver de monitorización ejecutándose de forma nativa y fuera del contenedor, siendo únicamente las aplicaciones de monitorización las que se encuentran virtualizadas y acceden a los dispositivos virtuales mediante mapeos a ficheros del driver nativo. Los resultados de estos test se muestran en la Figura 2.8, donde se observa el mismo efecto tanto en DPDK como en HPCAP40vf: Para paquetes de tamaño menor ambos tienen un rendimiento peor al ejecutarse en un contenedor Docker, sin embargo, el rendimiento mejora en función del tamaño de paquete a una velocidad mayor al ejecutarse en docker con respecto a KVM. No obstante, a pesar de que este efecto parece favorable con docker, al aumentar el tamaño del paquete a 256-Bytes o superior, ambos métodos de virtualización nos permiten capturar el 100% del tráfico sin perdidas.

Almacenamiento de tráfico

Una vez que hemos probado que los drivers tienen la capacidad de recibir el tráfico a una tasa razonable y que los medios de almacenamiento también son capaces de escribir a tasa de línea, es posible realizar pruebas que involucren la captura y almacenamiento simultáneo. El montaje es igual tanto para Docker

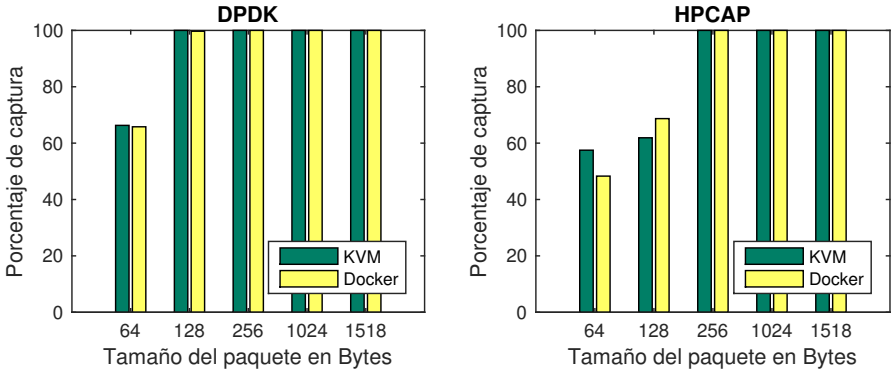


Figura 2.8. Comparación de porcentaje de captura con DPDK y HPCAP40vf en una máquina virtual KVM y un contenedor Docker.

como para KVM: Un RAID-0 software es montado utilizando 5 discos NVMe, ya que tal y como se reportó en la Figura 2.7 son suficientes para la captura. Nótese que en este contexto el driver del sistema de ficheros, del RAID y del propio NVMe se encuentran virtualizados en KVM mientras que en Docker se ejecutan de forma nativa sin ese sobrecoste. Después, el generador de tráfico introduce una traza con tráfico real con la que poder medir el ancho de banda de captura de nuestra VNP.

La aplicación de captura utilizada para medir HPCAP40vf es hpcapdd. Esta aplicación copia los datos en bloques desde el buffer intermedio a un fichero. Esta aplicación hace uso del RAID y el sistema de ficheros expuesto por el sistema operativo. Esto permite una solución más homogénea y compatible que recurrir a SPDK[9], el cual, utilizando los 11 discos NVMe paralelo de la máquina de medidas, alcanza una tasa de escritura de 100 Gbit/s. No obstante, dado que nuestro caso de uso se centra en los 40 Gbit/s y hemos confirmado que el software común de Linux alcanza esta tasa no es necesario recurrir a él.

En el caso de DPDK, existen algunas herramientas públicas de captura prometedoras. No obstante, estas herramientas han sido diseñadas para tomar una muestra de la red durante un tiempo pequeño y no sostenido a 10 o 40 Gbit/s. Una de las opciones a tener en cuenta es Flowscope [78]. En ese trabajo, los autores dicen alcanzar una tasa de hasta 100 Gbit/s. No obstante, por diseño, su arquitectura no puede sostener esta tasa de captura ya que solo pueden volcar cierto número de paquetes al detectarse una anomalía y nuevamente no de forma continuada. Existen otras opciones basadas en DPDK como dpdkcap [79], pero su código está obsoleto y sin mantenimiento, lo que impide utilizarlo con un software

[9] http://www.spdk.io/

actualizado de DPDK y, por ende, que soporte la tarjeta que estamos utilizando. Por este motivo se desarrolló una herramienta de DPDK de captura[10] que permite volcar en PCAP a una tasa de hasta 40 Gbit/s.

		% DE ALMACENAMIENTO	
Traza	**Tasa de transmisión**	**DPDK**	**HPCAP40vf**
CAIDA	39.80 Gbit/s	99.5%	100.0%
UAM	39.83 Gbit/s	100.0%	100.0%

Tabla 2.5: Rendimiento al capturar y almacenar dos tipos diferentes de tráfico.

La Tabla 2.5 muestra los resultados obtenidos con dos trazas de tráfico diferentes: Una obtenida por el grupo CAIDA [77] y la otra es una muestra de tráfico de los laboratorios docentes de la Escuela Politécnica Superior de la Universidad Autónoma de Madrid. La media del tamaño del paquete es de 910 y 787 bytes respectivamente. Los resultados de esa tabla han sido obtenidos en KVM, se excluyen los resultados de Docker ya que la diferencia es despreciable. DPDK solo captura sin pérdidas la traza obtenida en la UAM. Sin embargo, HPCAP40vf es capaz de capturar todos los paquetes en ambos casos sin pérdidas, lo que muestra que de hecho es posible monitorizar y almacenar el tráfico de una red a 40 Gbit/s utilizando el concepto de VNP. Esta diferencia de resultados era la esperada. Tal y como se explicó anteriormente, HPCAP40 y HPCAP40vf han sido orientados a la captura y almacenamiento paquetes de forma que se realice una copia extra para poder alinearlos en memoria. Esto afecta al proceso inicial de captura, cuando el tráfico es descartado o en el caso de que una aplicación no necesitase volcar los datos a un fichero. No obstante, es el caso idóneo para una aplicación en la que unos de sus objetivos sea almacenar los datos a disco.

[10] `https://github.com/hpcn-uam/DPDK2disk`

2.2. Monitorización Activa

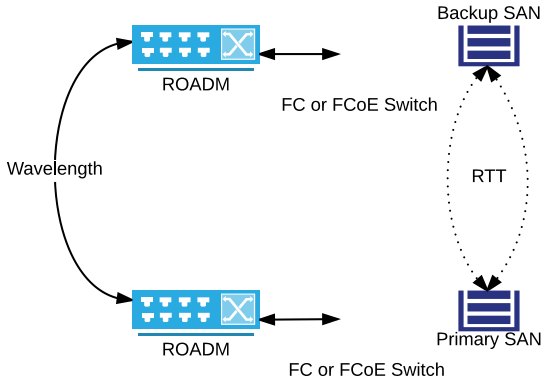


Figura 2.9. Escenario de replicación en un SAN

Uno de los casos de uso más claros de la monitorización activa es la medición de latencias. Realizar medidas de latencia precisas punto a punto en redes ópticas es de vital importancia para la evaluación del rendimiento de sistemas que se encuentran distribuidos entre diferentes centros de datos como, por ejemplo, los Storage Area Network (SAN). En la Figura 2.9 se muestra un escenario de replicación común entre dos centros de datos. Para poder realizar una escritura en esta arquitectura es necesario que los datos sean almacenados tanto en el SAN primario como en el SAN de backup simultáneamente, obligando a éste a confirmar de forma explícita que la información ha sido almacenada. Esto causa que la latencia de escritura esté restringida al tiempo de ida y vuelta o Round-Trip Time (RTT), entre el SAN primario y el de backup. Para acelerar las operaciones de escritura, es común reservar una fibra oscura o una longitud de onda únicamente destinada a comunicar el SAN primario con el de backup, de forma que las operaciones entre ambos no se vean en ningún caso interferidas por otro tráfico. Este escenario es común en aplicaciones donde deban ejecutarse transacciones críticas, como autorizaciones de uso de tarjetas de crédito, que además de requerir un tiempo de espera corto en la respuesta, requieren que todas las operaciones sean síncronas y transaccionales.

Siendo el RTT el tiempo de ida y vuelta entre el SAN primario y el secundario, y N el número de operaciones que se pueden ejecutar en paralelo en la base de

datos primaria, observamos que el ancho de banda, ρ, en operaciones de escritura por segundo está acotado superiormente de la siguiente forma:

$$\rho < \frac{N}{RTT} \tag{2.1}$$

Normalmente, los SANs replicados síncronamente no están localizados demasiado separados el uno del otro para evitar retardos especialmente grandes. Por ejemplo, un SAN primario y su backup pueden estar ubicados en la misma ciudad a una distancia de unos 10 kilómetros, con un RTT aproximado de unos $100\mu s$. Supongamos en este ejemplo, que $N = 10$ es el número de operaciones que la base de datos puede ejecutar en paralelo, lo que nos permitiría tener una tasa de 100,000 escrituras por segundo. Si el RTT se incrementase en tan solo $10\mu s$, el sistema pasaría a procesar apenas 90,909 operaciones de escritura por segundo, casi 10,000 operaciones menos, una cantidad muy significativa $\sim 10\%$. Estas pérdidas se deben al hecho de que la Ecuación 2.1 es hiperbólica, y pequeños incrementos en el denominador tienen un enorme impacto en el resultado.

La motivación anterior muestra que una estimación precisa del RTT adquiere una importancia fundamental a la hora de evaluar el rendimiento de los SANs. Existe la posibilidad de que el operador de red cambie la fibra óptica a una ruta más larga y el aumento resultante de la latencia, aunque se encuentre en la escala de microsegundos, tiene un gran impacto en el rendimiento final. Por lo tanto, la monitorización continua de la latencia a una resolución de microsegundos o superior es necesaria.

Para poder medir la latencia con una granularidad tan fina, se pueden utilizar reflectómetros en los segmentos puramente ópticos. Sin embargo, estos reflectómetros no pueden medir la latencia producidas por el equipamiento intermedio, como los switches de los SAN (de tipo Fibre Channel (FC), Fibre Channel sobre Ethernet (FCoE) o Internet SCSI (iSCSI)) que forman parte de la comunicación entre los extremos. Una forma de realizar la medida, incluyendo los saltos intermedios, son los trenes de paquetes [22, 80], ya que pueden ser enviados desde un extremo, y reenviados de vuelta al origen desde el segundo extremo, pudiendo así estimar el RTT. Este método es tan sencillo que puede ser implementado tanto a nivel hardware como software.

No obstante, una implementación dedicada basada en hardware a velocidades multi-gigabit es muy cara. Tal y como vimos en la Sección 2.1, actualmente existen soluciones software que ejecutando sobre hardware de propósito general pueden

obtener resultados similares al hardware a un coste menor. Por ese motivo, decidimos poner a prueba las soluciones software existentes. Volviendo al caso de uso de un despliegue SAN con escritura síncrona, es importante tener en cuenta que los controladores utilizados en estos despliegues normalmente utilizan un Unix o incluso Linux como sistema operativo. Éste es el entorno ideal para ejecutar un software de medida en los propios controladores, y mantener una monitorización continua sin necesidad de utilizar ningún hardware añadido. Esta solución permitiría a los gestores del SAN recibir alertas en caso de se observase un incremento en la latencia.

El objetivo de esta sección es analizar y diseñar una aplicación capaz de medir latencias en redes ópticas (con precisión de cientos o incluso decenas de nanosegundos) en software, y demostrar su viabilidad en futuros desarrollos siempre y cuando se tenga en cuenta que: (I) El software presenta características no deterministas debidas al encolamiento de paquetes y (II) el reloj clásico del sistema no posee una exactitud determinista, lo que complica el proceso de marcado temporal de paquetes.

2.2.1. Soluciones previas

Dentro de los sistemas de monitorización activa existen ciertas herramientas que se han convertido, dependiendo del entorno y contexto, en aplicaciones por defecto e imprescindibles. Algunos de ellas son iPerf[11], Aria2[12], o JMeter[13]. Cada una de estas herramientas está destinada a tomar métricas a un nivel diferente (red y aplicación). El rendimiento de estas herramientas está muy ligado al sistema operativo y a la pila de red que éste implemente. Es importante tener en cuenta que esta pila no es determinista en los sistemas operativos comunes como Linux, Windows o Mac. Como consecuencia, el tiempo que tarda cualquier función que haga uso del Kernel, lo que incluye las funciones recibir o enviar un mensajes y paquetes, no está acotado. Existen sistemas operativos (los denominados Sistemas Operativos en Tiempo real como Linux RT[14]) que intentan garantizar los tiempos de respuesta de las funciones, no obstante, tampoco son 100 % deterministas, pues los efectos de encolado de paquetes e interrupciones afectan igualmente.

[11] `https://github.com/esnet/iperf`
[12] `https://aria2.github.io/`
[13] `http://jmeter.apache.org/`
[14] `https://rt.wiki.kernel.org`

Como resultado, que el tiempo de llamada a función no esté garantizado por el sistema operativo tiene un severo impacto a la hora de realizar medidas de tráfico, en especial aquellas que necesiten mediciones de tiempo precisas como las latencias. Esto ha causado una enorme crítica hacia el uso de software como método para medir tiempos de forma precisa, delegando el trabajo en soluciones hardware deterministas, por ejemplo, FPGAs [81–84]

La primera solución por parte de la comunidad software para solucionar el problema de las mediciones activas fue crear *pktgen* [85]. *Pktgen* está diseñado como un módulo del Kernel, por lo que tiene permitido acceder directamente a las colas de las interfaces de red (NIC). Esto minimiza los sobrecostes, como los debidos a cambios de contexto entre el Kernel y el usuario, o copias intermedias de paquetes entre nivel de usuario y Kernel. Por tanto, *pktgen* tiene un rendimiento y fiabilidad superiores a sus hermanos basados en sockets estándar, como los citados al inicio de esta sección. No obstante, esto no es suficiente. Funciones básicas como un simple `printk` (función utilizada para imprimir en los logs del Kernel) tarda en ejecutarse un tiempo no determinista, así como otros componentes del Kernel que obstaculizaron a *pktgen* a alcanzar un máximo de 1 Gbit/s en sus implementaciones iniciales. A pesar del duro esfuerzo de la comunidad, y de increíbles mejoras en el rendimiento, *pktgen* sigue encontrándose bastante por debajo de los 10 Gbit/s [86].

Para poder obtener una solución lo menos variable posible, es necesario un software capaz de aislarse en unas CPUs determinadas, evitando cualquier cambio de contexto con otras aplicaciones o con el propio Kernel. Tal y como se ha explicado previamente en la Figura 2.1.1, DPDK cumple con todas estas ideas, a la vez que permite utilizar casi cualquier tarjeta de red moderna[15]. Existen además trabajos en los que se han diseñado equipamiento de red virtual bajo DPDK asegurando que se minimiza la latencia frente a otras soluciones software [87–90].

En este aspecto, Moongen [91] destaca como posible solución capaz de inyectar paquetes en la red, ya que ha sido implementado utilizando DPDK. Los creadores aseguran que utilizando scripts LUA son capaces de realizar mediciones a nivel de nanosegundo. No obstante, este nivel de precisión lo logran utilizando las capacidades de marcado de paquetes de las tarjetas de red, lo cual, lamentablemente, se encuentra limitado a algunas NICs y bajo ciertas restricciones, por ejemplo, que el paquete sea de tipo PTP en las tarjetas de Intel. Aunque podrían ser utilizados

[15] `http://dpdk.org/doc/nics`

este tipo de paquetes a la hora de realizar nuestras mediciones de latencia, restringir la medición de una red a únicamente un tipo de paquete impediría realizar medidas en una red en donde por defecto y seguridad este tipo de paquetes se encontrasen bloqueados por uno o varios de los equipos de red.

Por este motivo, el sistema de medición debe acomodarse a los requerimientos de seguridad de la red y simular diferentes tipos y categorías de tráfico (cada una con distintos perfiles de QoS). Volviendo al caso de uso de las SANs, en un enlace entre dos SAN que haga uso de FCoE, ningún paquete UDP o PTP podrá ser transmitido ya que FCoE es un protocolo de nivel 3. No obstante, esto no significa que no se pueda usar FCoE para medir latencias, ya que dispone de campos reservados en los que ubicar un posible timestamp. La única restricción en esta situación es una herramienta capaz de generar cualquier tipo de paquete y ubicar el timestamp en un lugar específico.

2.2.2. Inyección de paquetes con DPDK

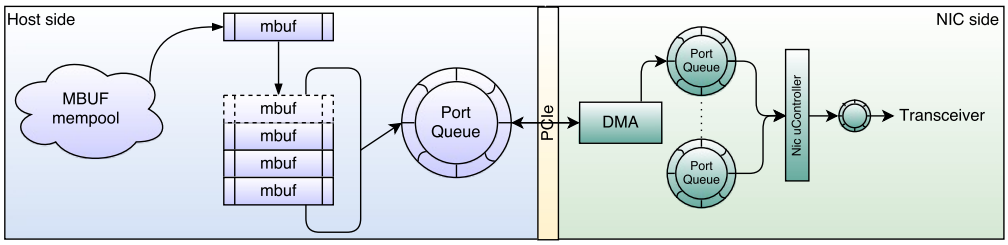


Figura 2.10. DPDK workflow

Para poder desarrollar un sistema preciso sobre DPDK es necesario profundizar un poco más en su funcionamiento, con respecto a la explicación en la Figura 2.1.1. En la Figura 2.10 se muestra un esquema detallado del flujo de *mbufs* en el proceso de transmisión de paquetes y todas y cada una de las colas software y hardware que debe atravesar.

Para poder enviar un paquete a la red, es necesario reservar su memoria asociada y su *mbuf*. Típicamente, y para ahorrar tiempo y llamadas al sistema, es necesario recurrir a los denominados *mbufpool*s, es decir, grupos de *mbuf*s prereservados. Cada *pool* es reservado en el nodo *NUMA* adecuado al arranque de la aplicación. Una vez el contenido del paquete ha sido copiado a la región del *mbuf* correspondiente, es necesario rellenar los metadatos, es decir, la longitud del paquete a transmitir y el puerto por el que se debe transmitir.

DPDK funciona por *batches* o grupos de paquetes. Por supuesto, la API de DPDK permite que el tamaño de un grupo de paquetes consista en un único paquete, no obstante, el hardware no tiene por qué aceptarlo, y podría forzar el encolamiento internamente hasta que este considerase la transferencia lo suficientemente óptima. Dado que el número de paquetes de un *batch* afecta al rendimiento, es común utilizar 144 *mbufs*, ya que suele ser el número recomendado por Intel y empíricamente es el óptimo también en otros fabricantes.

Cuando el puerto físico de una tarjeta NIC es inicializado, es obligatorio instanciar al menos dos colas software: una para transmisión y otra para recepción, independientemente de que no sea necesario utilizar alguna de las dos. DPDK define una cola como un buffer en anillo con un número finito de punteros a *mbufs*. Estos punteros pueden ser leídos y escritos por el hardware de la NIC, y utilizarlos para enviar los paquetes almacenados en ellos o escribir los paquetes recibidos.

La transmisión a nivel PCI implica cierta complejidad si se busca maximizar el ancho de banda y minimizar la latencia. Para mitigar los efectos causados

por PCI, la tarjeta utiliza un pequeño anillo interno en donde se almacenan de forma provisional los paquetes que previamente se encontraban en la memoria principal. Desde este anillo interno, los paquetes son preprocesados uno a uno secuencialmente por el controlador de la NIC. Este preproceso puede incluir pequeñas manipulaciones en los paquetes, como la generación del CRC, cálculo del checksum IP, etc.

Una vez el controlador de la NIC ha procesado el paquete, éste se encuentra listo para ser transmitido al medio físico. No obstante, dadas las particularidades de los medios de transmisión es común que el paquete quede retenido en una pequeña cola nuevamente. Por ejemplo, en 10 Gigabit Ethernet, la NIC utiliza transceptores SFP+ cuyo medio de comunicación con el chipset de la NIC es el protocolo *10 Gigabit Media Independent Interface* (XGMII). Este protocolo no permite que un paquete se transmita arbitrariamente en cualquier momento, ya que es necesario cierto alineamiento de bit. Este alineamiento puede hacer cambiar el conocido *interframe gap* de Ethernet que, si bien en media debe ser 12 Bytes, puede llegarse a reducir a 5 Bytes según el estándar [49]. Esto implica, que aun a pesar de lograr un sistema software completamente determinista, existen colas y factores a nivel hardware que pueden causar pequeñas variaciones e indeterminismos, aunque estos se encuentren en torno a los nanosegundos.

Limitaciones de DPDK en la transmisión

DPDK como motor de gestión de paquetes presenta algunas limitaciones, en parte por su diseño software y en parte debido a la implementación hardware de las tarjetas de red. Una de las limitaciones que encontramos con las tarjetas de 40 GE Intel XL710, es el tamaño mínimo del *burst* de paquetes que se puede enviar o, dicho de otra manera, el número mínimo de paquetes que se pueden transmitir si la cola de transmisión se encuentra vacía. Este tipo de limitaciones están relacionadas con optimizaciones del ancho de banda de PCI, por lo que realmente el número mínimo de paquetes no es elevado. En el caso concreto del chipset *XL710* es tan solo de 4 paquetes. Es interesante aun así tenerlo en cuenta, pues a nivel software tanto en transmisión como en recepción estos 4 paquetes serán transmitidos y recibidos simultáneamente, lo que complica parcialmente obtener mediciones precisas. Otro factor a tener en cuenta es la latencia que se produce desde que se emite la orden, hasta la copia de los paquetes a la tarjeta. Para DPDK transmitir un paquete consiste en copiar los *mbufs* a la cola correspondiente. No

obstante, es la NIC la que debe acceder a esa memoria y pueden pasar varios nanosegundos entre la copia por parte de DPDK, y la copia del paquete por parte de la tarjeta hasta su cola interna. Una vez se ha copiado, volverán a pasar ciertos nanosegundos hasta que el controlador y el serializador transmitan de forma efectiva el paquete por el medio físico. Otro dato a tener en cuenta es el agrupamiento de grupos. Si bien un primer grupo puede tardar en enviarse, si la tarjeta se encuentra de forma activa copiando el grupo anterior y en ese momento DPDK inserta en la cola un nuevo grupo de paquetes, este segundo grupo no tendrá la latencia inicial del primero, ya que la tarjeta simplemente lo verá como un grupo más grande y, por tanto, desaparece el tiempo entre ambos grupos (intencionado o no por parte de la aplicación). Dado que el tiempo que tarda la NIC desde que se encola un grupo de paquetes hasta que se transmite (principalmente si la cola llega a vaciarse) no es constante, pueden producirse importantes sesgos en la medida alcanzando el nivel de los microsegundos y, por ende, afectando severamente la medida.

La segunda limitación de DPDK es el hecho de que todas las colas intermedias descritas anteriormente introducen retardos no deterministas. Cuando el ancho de banda de la red se incrementa, el tiempo entre paquetes disminuye, introduciendo pequeños retardos aleatorios que hacen oscilar el número de paquetes por segundo en breves espacios de tiempo y, por ende, añaden ruido a la medida. La única forma de superar esta limitación es enviar un número suficiente de paquetes de trenes consecutivos, y realizar una estadística sobre cada una de las medidas.

La tercera limitación de tomar medidas de latencia en software no afecta en exclusiva a DPDK pues está relacionada con la precisión del reloj del sistema. Una funcionalidad tan básica como la obtención del tiempo actual varía en precisión, exactitud y coste computacional entre diferentes CPUs y versiones del Kernel de Linux. Para obtener una medida temporal, del orden de los cientos de nanosegundos, es necesario utilizar el mínimo de nanosegundos posible (al menos un orden de magnitud menor). La mejor manera de lograrlo es utilizar el contador de ciclos de la CPU (*Time Stamp Counter* o *TSC*), ya que tiene una resolución a nivel de decenas de nanosegundos. No obstante, este método es solo útil si la CPU soporta *Invariant TSC*, es decir, el contador de ciclos es compartido entre todas las CPUs y cores, e independientemente de si se apagan o cambian su frecuencia de forma individual, el contador siempre se actualizará a la frecuencia del más rápido.

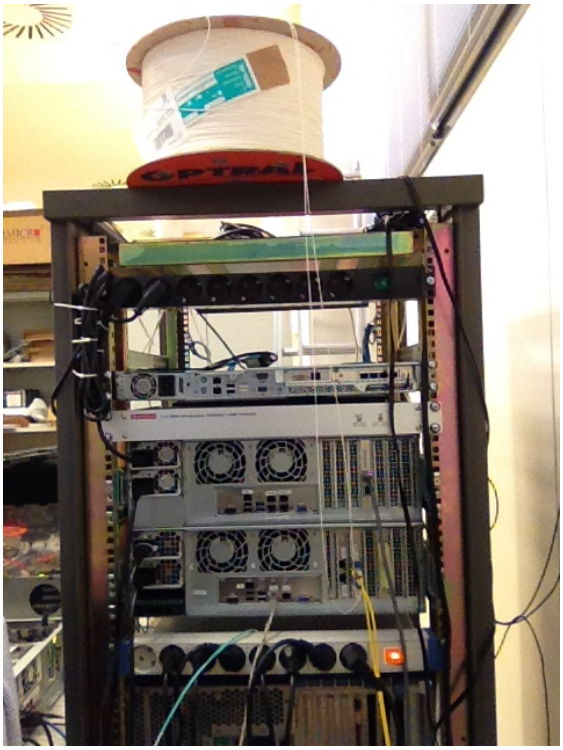


Figura 2.11. Bucle de fibras utilizadas en las pruebas. El rollo de fibra blanco sobre el rack tiene 2025 metros de largo.

Configuración experimental

Para poder evaluar de forma empírica el impacto de estas limitaciones se decide realizar pruebas controladas en escenarios en los que la latencia real se mantenga constante y sea de antemano conocida. Para ello, se conectan fibras de diferentes longitudes en bucle (en la Figura 2.11 se puede observar el montaje experimental).

Para esta prueba utilizamos un único servidor con dos procesadores Intel Xeon E5-2630 a 2.30 GHz, cada uno con 16 GB de RAM a 1333 MHz repartidos en 4 tarjetas para aprovechar los beneficios del quad-channel. La interfaz de red es una Intel Ethernet de 10 Gigabit con el chipset 82599ES. El sistema operativo es un Linux CentOS 7.3 y DPDK 17.05 como motor de transmisión y recepción de mensajes.

Uno de los efectos a tener en cuenta en software, y que de forma natural son menos relevantes en hardware, es la longitud del paquete a la hora de medir la latencia. En hardware, basta con que el marcado temporal esté posicionado relativamente al comienzo del paquete para que éstos sean los primeros bytes en ser

transmitidos y recibidos. En software, en cambio, no existe el concepto de enviar medio paquete y como hemos explicado con anterioridad, el concepto de un único paquete tampoco es muy recomendable en DPDK. En consecuencia, dado el índice de refracción de la fibra (n), la longitud de la fibra en metros (l), la velocidad de la luz en el vacío en metros por segundo (c_0), la tasa de transmisión del enlace en bits por segundo (r) y la longitud del paquete en bits (b), el delay teórico producido en la fibra (d) puede ser calculado fácilmente como:

$$d = d_{prop} + d_{tx} = \frac{n \cdot l}{c_0} + \frac{b}{r} \tag{2.2}$$

en donde el primer término se refiere al tiempo de propagación en el cable (d_{prop}), y el segundo término al retardo debido al tiempo de transmisión (d_{tx}).

Utilizando la Ecuación 2.2, en una fibra de 2 metros de largo con un índice de refracción típico de 1.4444 transmitiendo a una tasa de 10 Gbit/s, el retardo del paquete más pequeño posible (60 Bytes a nivel Ethernet excluyendo CRC, preámbulo y tiempo inter-paquete (inter-frame gap)) sería de 96 nanosegundos, y 1262 nanosegundos en el caso del paquete de mayor tamaño (1518 Bytes a nivel Ethernet incluyendo protocolo VLAN). Por supuesto, si contamos con tamaños de paquete superiores (los conocidos “Jumbo Frames”), los retardos serán superiores.

2.2.3. Principios de diseño para mediciones de latencia de alta resolución

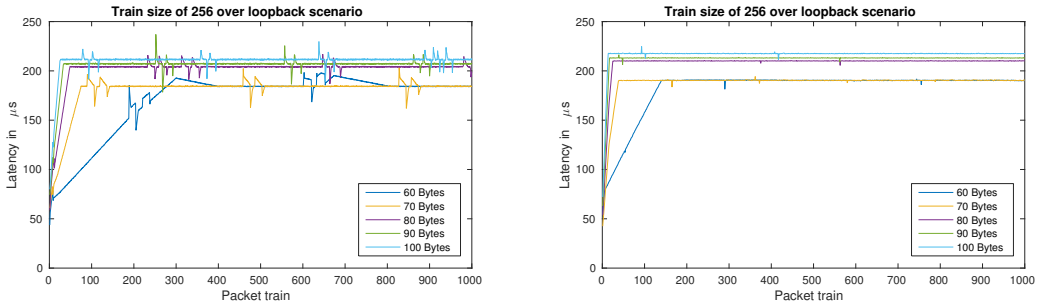
Debido a las limitaciones de DPDK mencionadas hasta ahora, las herramientas disponibles en el momento de la realización de las pruebas, como DPDK-pktgen[16] o moongen [91], no se encontraban lo suficientemente afinadas para poder realizar este tipo de medidas de manera adecuada.

Por este motivo, se decide crear una herramienta propia, llamada DPDK-LatencyMetter, y publicarla de forma abierta[17]. Esta nueva solución, funciona con cualquier modelo de tarjeta y fabricante, al contrario que moongen [91], y permite obtener las medidas con la precisión necesaria para el caso de uso propuesto anteriormente. A continuación, se muestran algunos detalles técnicos relacionados con el diseño de DPDK-LatencyMetter.

[16] https://github.com/pktgen/Pktgen-DPDK
[17] https://github.com/hpcn-uam/iDPDK-LatencyMetter

Aislamiento de CPUs

Una de las funciones del sistema operativo es distribuir los procesos de forma equilibrada entre las CPUs, compartiendo los recursos de la manera más eficiente posible. No obstante, para minimizar efectos aleatorios, todas las medidas deben ser realizadas en determinados cores que deben estar aislados de cualquier otro proceso del sistema. Este tipo de aislamiento se puede lograr utilizando el parámetro del Kernel de Linux `isolcpus`. No obstante, aunque no exista un segundo proceso asignado al core en el que se ejecutan los procesos de medida, eso no impedirá al Kernel interrumpir al proceso de forma periódica (los llamados "ticks") para comprobar si otro core ha asignado un proceso al core recién interrumpido. Por este motivo, el Kernel de Linux debe ser compilado explícitamente con la opción `tickless`. Este modo del Kernel, no inhabilita los ticks por completo, ya que son fundamentales (se ejecutarán siempre en el core 0 de cada CPU física) pero no en el resto de cores. Los cambios de proceso si son necesarios, son realizados cuando el proceso hace una llamada al sistema.



(a) Test realizado en un Linux con ticks periódicos habilitados, sin *isolcpus* y en modo *powersave* (modo por defecto)

(b) Test realizado en un Linux sin ticks periódicos e *isolcpus* activo. La CPU se encuentra en modo *performance*.

Figura 2.12. Latencia medida para 1000 trenes consecutivos, cada uno formado por 256 paquetes.

Estos efectos son claramente apreciables cuando enviamos 1000 trenes de paquetes consecutivos, y medimos la estimación de la latencia de cada uno de ellos. El la Figura 2.12a se muestra este escenario con una longitud de tren de 256 paquetes, sin ningún tipo de aislamiento (ni *tiks* ni *isolcpus*). Es interesante observar unos picos disruptores en la medida con una frecuencia constante para cada tamaño de paquete. Esto se debe al hecho de que un tren con paquetes de tamaño grande requiere más tiempo en enviarse o, en otras palabras, el tiempo entre mediciones de latencia es superior. Este es el motivo por el que los trenes con

paquetes más grandes sufren con mayor frecuencia anomalías en la medida, y los trenes de paquetes menores con menor frecuencia. Estas anomalías desaparecen en cuanto las opciones `tickless` e `isolcpus` se encuentran activas (tal y como se muestra en la Figura 2.12b).

Políticas de ahorro energético

Las políticas de ahorro energético afectan de forma directa al reloj de la CPU, y por ende al rendimiento y el número de MIPS (*Millones de Instrucciones Por Segundo*) que puede alcanzar. Es interesante destacar que el número de MIPS que puede ejecutar el procesador no tiene un impacto relevante en las medidas de latencia, pero si lo tiene en las de ancho de banda. Esto se debe a que las medidas de latencia no necesitan que los paquetes se produzcan a una tasa determinada, mientras que las medidas de ancho de banda son directamente influenciadas por la tasa de generación de paquetes. En cambio, el número de paquetes que se pueden generar y encolar por segundo dependen directamente de la frecuencia del núcleo de CPU en el que se encuentren y la cantidad de MIPs que este pueda ejecutar, al igual que otros factores, como la implementación del driver de la tarjeta o el ancho de memoria.

Cuando el número de MIPS es suficiente para alcanzar la máxima capacidad del enlace, ambas métricas (ancho de banda y latencia), se vuelven estables en algún momento, independientemente de si la frecuencia aumenta. No obstante, el tiempo necesario para alcanzar este estado estacionario depende directamente de los MIPS y el número de paquetes por segundo necesarios para alcanzarlo, que a su vez depende del tamaño de paquete, tal y como se puede observar en las rampas crecientes de la Figura 2.12.

Efectos del tamaño de paquete

Usando el método de trenes de paquetes más simple, se observa una correlación entre el tamaño del paquete y la medida de la latencia. Para entender mejor este efecto, se realizó un experimento enviando 100 trenes de paquetes con diferentes tamaños de paquete. Por razones de visibilidad, se escoge un tamaño de paquete en un intervalo de entre 60 y 6000 Bytes a nivelEethernet (excluyendo CRC, preámbulo e inter-frame gap) y se representan los resultados en términos de

índices de dispersión de intervalos, IDI, definido como la varianza de cada prueba dividida entre su media, como se muestra en la Ecuación 2.3.

$$IDI_{retardo} = \frac{\sigma^2_{retardo}}{\mu_{retardo}} \tag{2.3}$$

Al realizar este tipo de medidas con sistemas no deterministas existe la posibilidad de obtener mediciones extremadamente elevadas, por ello y por razones de visibilidad, los valores por encima de 1.5 veces el rango intercuartílico han sido considerados como outliers y por lo tanto eliminados de la muestra. Uno de los efectos observados, es que cuando la longitud de los paquetes se acerca a tamaños concretos, se produce un pico en la varianza de la medida. Potencialmente este efecto se debe a pequeños desalineamientos en las transferencias de PCIe. PCIe trabaja como una red conmutada de paquetes (conocida como *Transaction Layer Packet* (TLP)) en donde los paquetes siempre tienen un tamaño fijo que se negocia en el arranque del dispositivo. Dependiendo del fabricante y modelo de la tarjeta de red, este valor puede variar. Cuando una transferencia por PCIe es ligeramente superior a un múltiplo del tamaño del TLP, se produce un desperdicio en el ancho de banda utilizado por PCIe. Por ejemplo, si el TLP tiene el tamaño fijado a 64 Bytes, un paquete de red de 65 Bytes necesitaría utilizar 2 paquetes TLP de tamaño 64, produciendo en este caso un desperdicio cercano al ~ 50 %.

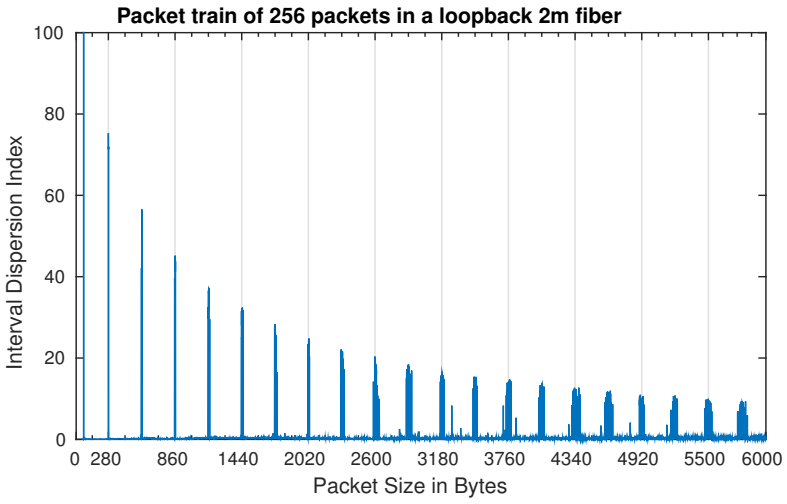
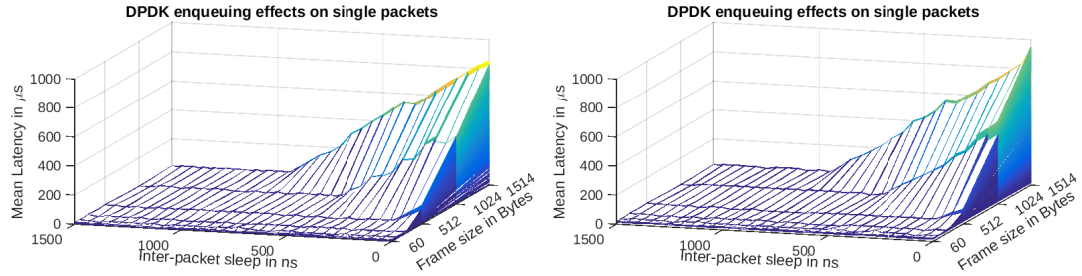


Figura 2.13. El índice de la dispersión de intervalos (IDI) sobre dos metros 2 de fibra. El test se realizó utilizando diferentes tamaños de paquete con trenes de 256 paquetes de longitud.

En la Figura 2.13 se observan los picos de varianza producidos cuando los paquetes del tren tienen más de 65 Bytes de longitud. Después de este efecto particular y empezando en 280 Bytes, por cada 288 Bytes, se produce un incremento en el IDI. Este efecto cambia en intensidad y periodicidad entre diferentes máquinas y tarjetas de red, pero es un efecto presente en todos los entornos en los se ha probado, y siempre presenta un efecto constante para una combinación de hardware y software determinada. En el test de la figura, también se observa que la media y la mediana de la latencia varia ligeramente por cada 32 Bytes, creando un perfil de sierra en las medidas. Es posible que ambos efectos están relacionados, ya que el aumento de la varianza producido cerca de los 288 Bytes, que es 9 veces 32. Al replicar este experimento con una CPU distinta, y otra tarjeta de 10Gbit de Intel pero con chipset distinto, se observa que el efecto se produce cada 320 Bytes, que en este caso, es exactamente 10 veces 32.

Efectos de encolado



(a) Bucle de fibra de 2 m. **(b)** Bucle de fibra de 2 Km.

Figura 2.14. Efectos de encolamiento variando el tamaño y el tiempo entre paquetes.

Para evaluar los efectos del encolamiento de la medida, se decide realizar pruebas de transmisión con paquetes individuales. La Figura 2.14 muestra las mediciones de latencia para diferentes tamaños de paquete y un tiempo entre paquetes, éste último añadido virtualmente en dos escenarios distintos: bucles de fibra de 2 y 2025 metros. La única diferencia apreciable en la medida entre ambos escenarios es el valor mínimo: $4\mu s$ en Figura 2.14a y $13\mu s$ en Figura 2.14b. Claramente observamos que se produce un error en la medida (números mucho más grandes de lo esperado), tanto si se incrementa el tamaño de los paquetes como si se disminuye el tiempo entre paquetes, ya que, en ambos casos, las colas de transmisión comienzan a llenarse más rápido de lo que se vacían.

En el caso de los paquetes pequeños, el tiempo de transmisión a 10 Gbit/s, se encuentra en torno a los 70 nanosegundos, y el coste de construir un único paquete y encolarlo, requiere más tiempo en nuestra CPU que el tiempo que necesita la tarjeta de red para transmitirlo. Por este motivo se cree que la mejor forma de obtener mediciones de latencia utilizando paquetes sueltos es que tengan el tamaño mínimo. Los casos en donde se producen sobreestimaciones de latencia están debidos tanto a efectos de encolamiento en DPDK, cómo a los sobrecostes del hardware subyacente, causando una tasa de transmisión variable en vez de la constante esperable. Ya que la tasa de llegada de paquetes a los subsistemas de transmisión es constante (y controlable mediante nano pausas), existen posibilidades de que los nuevos paquetes se encuentren con algún paquete previo en la cola, causando una pequeña latencia en la medida. A pesar de que los resultados con paquetes de 60 Bytes proporcionan tasas aparentemente estables, sigue existiendo una pequeña varianza que se puede apreciar en la gráfica anterior como ruido. Para asegurarnos de que dicha varianza es debida a efectos aleatorios del sistema operativo, y en ningún caso a un hipotético error conceptual o estructural, se puede analizar la caída de la media de la varianza según se incrementa el número de muestras.

Para este fin, se considera un grupo de N medidas, $X_1,\ldots,X_N$, independientes e idénticamente distribuidas como hipótesis nula, con una varianza σ^2 y una media μ. Como resultado, la media de la muestra

$$S = \frac{1}{N}\sum_{i=1}^{N} X_i \tag{2.4}$$

es un estimador insesgado de μ y

$$Var(S) = \frac{1}{N}\sigma^2 \tag{2.5}$$

Se han realizado mil pruebas enviando $N = 100$ paquetes muy separados en el tiempo ($15\mu s$), y calculado la correspondiente media S. Según Ecuación 2.5, la varianza decae logarítmicamente con respecto al número de muestras N que es equivalente a:

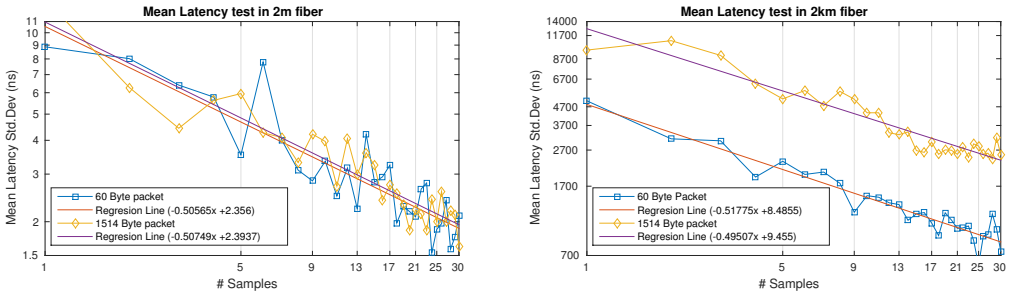
$$Log(Var(S)) = Log\left(\sigma^2\right) - Log(N) \tag{2.6}$$

también conocido como decrecimiento lineal con $Log(N)$.

En las Figura 2.15 se observa cómo se cumple la fórmula, para 2 metros (izquierda) y 2 kilómetros (derecha) de longitud de fibra respectivamente, y verifica

gráficamente la hipótesis nula de independencia. Por tanto, ya que la hipótesis era que la varianza se debía a latencias aleatorias del sistema operativo y del hardware, podemos concluir que la arquitectura es adecuada.

La misma figura puede usarse para buscar el número de veces que un tren de 100 paquetes debe ser repetido para obtener la varianza de la media (exactitud) deseada.



(a) Prueba sobre 2 metros. **(b)** Prueba sobre 2025 metros.

Figura 2.15. Convergencia de la media de la desviación estándar sobre una fibra utilizando medidas con paquetes unitarios.

En este punto podemos concluir que el uso de trenes de paquetes sin control (saturando la cola), conlleva un sesgo severo en las medidas realizadas con DPDK.

Calibración

Para tomar medidas precisas en DPDK, es necesario realizar una calibración para ajustar el software a las restricciones del hardware que utiliza, como la velocidad de la memoria, de la CPU, la versión del PCI y el número de líneas de la tarjeta, así como otros parámetros de bajo nivel que deben tenerse en cuenta.

Un ejemplo de cómo algunos detalles de bajo nivel pueden afectar en una medida que requiere precisión de centenas de nanosegunos, es el funcionamiento de la propia memoria RAM. Según la especificación del estándar DDR4[18], leer una línea de caché (64Bytes contiguos) tiene un retardo que puede oscilar entre 15 y 20 nanosegundos, dependiendo del módulo en el que se encuentre. Algunos procesadores pueden manejar hasta 6 canales de DDR4, lo que significa que son capaces de leer hasta 6 líneas de caché consecutivas en un único acceso, es decir hasta un total 384 Bytes en unos 15 o 20 ns. Además, el controlador de memoria de la CPU debe realizar órdenes de refresco periódicas sobre la memoria. Si este controlador no es lo suficientemente inteligente, podría darse el caso de que la dirección

[18] `https://www.jedec.org/document_search?search_api_views_fulltext=jesd79-4`

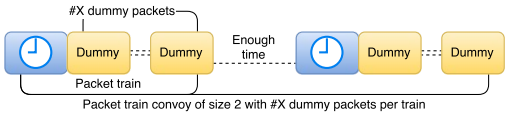


Figura 2.16. Convoyes de trenes de paquetes.

memoria que va a ser accedida, se encuentre en una región que esté realizando un refresco, lo que impediría la lectura de dicha dirección hasta que la memoria terminase el refresco. Este último efecto debería ocurrir con muy poca frecuencia, pero sigue siendo un factor a tener en cuenta.

En vista de lo anterior, podemos considerar la calibración como un elemento fundamental previo a la medida. El método más sencillo de realizar la calibración consiste en conectar en bucle una fibra a la interfaz de red que se desee utilizar para medir, y realizar algunas medidas de prueba. La longitud e índice de refracción de la fibra son necesarios para tener un valor teórico con el que comparar la medida. Mediante comparaciones del valor teórico con los valores medidos, el ruido de bajo nivel, como puede ser el tiempo de tránsito entre las colas de la NIC y PCIe, puede ser eliminado incluso si los efectos del encolado se encuentran en la escala de microsegundos.

2.2.4. Extensiones a las medidas de ancho de banda

Aunque hasta el momento este capítulo se ha centrado en las medidas de latencia, es posible extender las mismas ideas para medir el ancho de banda. En este caso, basta con mandar trenes de paquetes a máxima velocidad, de forma que el ancho de banda se calcularía en el receptor como la longitud del tren de paquetes en bits dividido por la duración de la transmisión.

No obstante, para poder asegurar que se mantiene la transmisión a máxima tasa, es necesario que la cola de *mbufs* llegue a un estado de saturación y, por tanto, la medida de la latencia se vea severamente distorsionada tal y como muestra la Figura 2.14. Una posible solución para estimar latencia y ancho de banda simultáneamente es utilizar pares de paquetes [92]. No obstante, esta solución no es adecuada para cualquier tipo de red. Por ejemplo, para un enlace agregado LACP de n fibras físicas, la medida con un par de paquetes nos indicaría un ancho de banda que podría ser hasta n veces menor del ancho de banda disponible. Para poder superar las limitaciones de los pares de paquetes, se propone una nueva metodología de medida capaz de medir ancho de banda y latencia simul-

táneamente aprovechando todas las capacidades del software. A esta metodología se le ha dado el nombre de *convoyes de trenes de paquetes*. La idea del convoy se centra en la transmisión de trenes consecutivos de paquetes a máxima velocidad para saturar las colas de transmisión. En este esquema, solo se añade una marca temporal a los primeros paquetes de cada tren para usarlos en la medida de RTT. Con este esquema, el primer paquete del primer tren del convoy no tendrá ningún efecto secundario ni de distorsión en la medida de la latencia, si es marcado temporalmente justo antes de encolar todo el tren (más allá de los problemas típicos del software). Por este motivo, solo un paquete del tren es usado para latencia mientras el resto es utilizado para medir el ancho de banda (ver Figura 2.16). Es muy importante que el marcado de paquetes se haga justo antes de enviar el `mbufarray` completo para acercar lo máximo posible el timestamp al momento de transmisión del paquete. Este método también proporciona una métrica de paquetes descartados, ya que, al no encontrarse la red completamente liberada, es posible que se produzcan perdidas de paquetes durante la medición. Marcando todos y cada uno de los paquetes podemos verificar en recepción si se ha producido desorden, y cuantos y cómo se han perdido.

En la Tabla 2.6 se muestran los resultados obtenidos con la metodología de convoyes de trenes de paquetes sobre las fibras de 2 metros y 2 kilómetros de largo. Los test se realizaron con 100 trenes, cada uno de 100 paquetes de longitud.

En este experimento, el ancho de banda de ambos escenarios debería ser el mismo, es decir, 9.844 Gbit/s para paquetes de tamaño 1518 Byte y 7.143 Gbit/s para paquetes de tamaño de 60-Bytes. Estos números teóricos coinciden bastante con los resultados empíricos obtenidos en la Tabla 2.6. Es importante tener en cuenta, que el ancho de banda útil teórico no son 10 Gbit/s, sino que depende del tamaño medio de los paquetes. Esto se debe a que al hablar de ancho de banda no incluimos bits que viajan por la red, como el CRC, el preludio o el IFG, los cuales, al transmitirse por cada paquete, tienen un peso mayor proporcionalmente cuanto más pequeño es el paquete transmitido. En términos de latencia, en una fibra de 2 metros, los resultados deberían ser de 144 ns para paquetes de tamaño 60 Bytes y 2.470 ns para paquetes de tamaño 1518 Bytes. Observando los resultados, es interesante comprobar que el error cometido varía en función del tamaño del paquete, y se confirman una vez más esos pequeños efectos que veíamos en las secciones anteriores. No obstante, el error cometido para un tamaño de paquete concreto es estable. Los resultados esperables para la fibra de 2 km son 10.057 ns y 12.383 ns para 60 y 1518 Bytes respectivamente. Tal y como se esperaba en

este experimento, el error cometido es equivalente al de la fibra de 2 m para un tamaño de paquete concreto.

Longitud de la fibra	Tamaño del paquete	Latencia estimada	Ancho de banda estimado
2 m	60 Bytes	582 ns	7.1 Gbit/s
2 m	1518 Bytes	6,685 ns	9.8 Gbit/s
2 Km	60 Bytes	10,510 ns	7.1 Gbit/s
2 Km	1518 Bytes	16,777 ns	9.8 Gbit/s

Tabla 2.6: Estimación del ancho de banda obtenido en un escenario en bucle utilizando convoyes de trenes de paquetes

Como parte negativa de esta metodología, se requiere mandar una cantidad enorme de paquetes, consumiendo todo el ancho de banda del enlace óptico durante un (breve) periodo de tiempo. Como cualquier otro método de medición activo, éste puede interferir con el tráfico en producción. Por este motivo este método debería ser utilizado en ciertas situaciones, y combinarlo con otros métodos de medición de latencias como el método basado en el envío de un único paquete (o trenes de paquetes pequeños) presentado anteriormente. Por ejemplo, medir la latencia es adecuado durante las horas puntas mientras que el ancho de banda solo deber ser medido en horas donde un tráfico puntual no sea crítico, pudiendo tener una vista diaria de todos los enlaces ópticos en uso.

2.3. Conclusiones

A lo largo de este capítulo se han presentado y analizados dos modelos de monitorización: Pasiva y Activa. Dentro de la monitorización pasiva se ha presentado el concepto de sonda de monitorización virtualizada (o en inglés VNP) utilizando tarjetas de red de 40 Gbit/s estándar. El modelo de monitorización asegura la posibilidad de mantener el ciclo observar-analizar-actuar que debe encontrarse presente en las redes 5G. Proceso que, por otro lado, parece fundamental para tener una red en condiciones óptimas. Gracias a las capacidades de cómputo de estas redes, la VNP puede ser desplegada con facilidad a lo largo de la red sin requerir un hardware específico, pudiéndose convertir tanto en una solución de monitorización permanente cómo en una solución lanzada bajo demanda siguiendo un modelo de monitorización bajo demanda. Tras explorar diferentes posibilidades de monitorización y virtualización de sondas, en la Sección 2.1 se muestra que

una arquitectura de monitorización que utilice hardware comercial no específico es posible tanto de forma nativa cómo utilizando virtualización de diferentes clases. No obstante, la monitorización basada en VFs presenta ciertas limitaciones que deben ser tenidas en cuenta por parte de los operadores de red para no causar sobrecargas y por ende pérdidas en los diferentes componentes virtualizados. En el caso de las tarjetas Intel de 40 GbE además considerar que una interfaz solo es capaz de transmitir por PCIe unos ~44 Gbit/s.

Dentro de la monitorización activa se ha explorado de forma breve el estado del arte de las tecnologías de monitorización activa, y se ha presentado una herramienta basada en DPDK para la realización de medidas activas, incluyendo de forma detallada los parámetros de DPDK que deben ser considerados y que influyen fuertemente dentro de la calidad de la medida. Finalmente, se ha propuesto una metodología propia y novedosa para la medición activa de enlaces ópticos de alta velocidad utilizando hardware de propósito general y un bajo coste, evitando recurrir a hardware de propósito específico y elevado coste.

3

Monitorización distribuida

El uso de la tecnología a lo largo del mundo unido al proceso de globalización, hace que la descentralización se convierta en un factor clave para el éxito de muchas empresas. Acercar los servicios al usuario final permite proporcionar una mejor calidad de servicio así cómo una menor latencia de red o un mayor ancho de banda. Aunque en general mejorar estos factores es importante para la mayoría de servicios, se vuelve un factor clave en algunos, cómo la comunicación de voz sobre IP, el vídeo en streaming o los videojuegos en tiempo real. Esta distribución del servicio impide el uso de aproximaciones clásicas de monitorización en donde unos pocos puntos de medida se concentran en un único analizador, ya que la información obtenida utilizando este método sería incompleta.

Monitorizar un sistema distribuido es una tarea cuya dificultad depende de forma directa del funcionamiento del servicio monitorizado. Si bien, este puede encontrarse dividido en diferentes ubicaciones geográficas a cientos o miles de kilómetros entre sí, si dicho servicio internamente puede describirse como un conjunto de subservicios independientes que no requieren de ninguna o muy baja comunicación entre sus nodos geográficos, es posible que el modelo de monitorización tradicional pueda ser aplicable, tratando cada uno de los subservicios como

sistemas independientes y cada uno de ellos con un sistema de monitorización dedicado. Desafortunadamente, esta aproximación no suele ser posible o se vuelve una solución incompleta y con elevados costes, ya que un servicio distribuido suele requerir de cooperación entre los diferentes puntos geográficos en mayor o menor medida, no pudiendo ser tratado como un conjunto de pequeñas entidades individuales sino como una única entidad dividida.

Por este motivo, es necesario utilizar un sistema de monitorización distribuido para poder analizar el comportamiento de esta clase de servicios distribuidos. El método de análisis de paquetes explicado en el Capítulo 2 para un sistema distribuido, requeriría la retransmisión de todos (o un subconjunto grande) los paquetes de red, lo cual hace completamente inviable esta aproximación en términos de volumen de transferencia de datos, capacidad de procesamiento y por ende, de costes. Por este motivo, los agentes de monitorización deben tener la capacidad de realizar un pre-análisis que reduzca la información y discrimine los paquetes o eventos relevantes de los irrelevantes así como agrupar y comprimir la información. En términos de red, podemos hablar de varios niveles de agregaciones –Por ejemplo a nivel de flujos, de superflujo, de subred, etc.– con las que generar registros así como gran diversidad de métricas sobre cada nivel de agrupación.

Estos registros pueden ser generados en formato binario (típicamente el formato usado para almacenar flujos de NetFlow o IPFIX) o en texto plano, siendo interpretable este último formato como un caso especial de *logs* de red. Los *logs* son registros generados por la mayoría de sistemas y aplicaciones durante su ejecución. Estos, proporcionan una trazabilidad de los múltiples tipos de eventos que han surgido permitiendo su posterior análisis de seguridad (ataques e intrusiones), errores de programación o de configuración, cuellos de botella, estado del equipo hardware subyacente entre otros muchos posibles elementos que de una forma u otra puedan ser relevantes para hacer seguimiento de la calidad del servicio y de los sistemas [93].

3.1. Loginson

Para poder realizar una monitorización basada en *logs* eficiente y usable, es necesario que los diferentes elementos que forman parte del sistema envien sus *logs* a un sistema (centralizado o distribuido) encargado de almacenarlos y analizarlos. Esta tarea puede parecer sencilla a ojos inexpertos pero puede acabar

fácilmente convirtiéndose en un verdadero reto ya que al alcanzar los cientos o miles de equipos enviando varios miles o decenas de miles de registros por segundo la capacidad de cómputo necesaria para almacenar y analizar los registros se dispara. Uno de los mayores problemas a la hora de analizar *logs* reside en la flexibilidad a la hora de definir su estructura de datos. Tanto en los registros binarios cómo en los de texto, los delimitadores de campos y las representaciones internas son completamente dependientes de la aplicación, lo que complica enormemente la extracción de dicha información en un sistema genérico pues es necesario aplicar diferentes mecanismos (cómo expresiones regulares en los *logs* de texto) para extraer la información relevante (tipo del evento, hora, estado de la CPU, etc.). Un ejemplo de posible log es el proporcionado por el conocido servidor web Apache, en el cual figura la IP de la petición, el usuario, el día y hora, la url, el resultado de la operación (200 OK) y el tamaño del archivo transmitido junto con el identificador del navegador web:

```
46.6.171.90 - root - [27/Aug/2019:14:35:38 +0200] "GET /grafana/ HTTP/1.1"200 8147 Mozilla/5.0 [...]"
```

Por norma general, un único registro no aporta información relevante a un administrador, a excepción de que dicho registro concreto presente una anomalía o evento de interés muy evidente cómo una intrusión de un usuario no autorizado. Por este motivo, la agregación y análisis de los logs es una parte fundamental para poder obtener información útil y legible de los mismos. A partir de esta necesidad se colaboró en el desarrollo de Loginson [94], un sistema de monitorización y seguimiento de Logs cuyo desarrollador principal fue *Carlos Vega* y el cual ha sido extensamente explicado en su tesis [95]. No obstante, esta herramienta ha sido incluida de forma muy resumida en este capítulo ya que la colaboración en su desarrollo generó experimentos e ideas que se han usado a lo largo de la tesis, en especial, motivó el desarrollo de *Wormhole* [96]. Por este motivo los detalles de implementación así como las pruebas de rendimiento no se incluirán en este capítulo. A su vez, no es posible tener en cuenta un sistema de monitorización integral sin incluir un método de monitorización a nivel de aplicación.

El objetivo de *Loginson* es el procesamiento y análisis de varios millones de líneas de log por segundo. Si consideramos una media de 300 Bytes por registro, un enlace de 10 Gbits se saturaría a un ratio de algo más de 3 millones de registros por segundo (teniendo en cuenta el encapsulado de red). Esto es el equivalente

a unos 30 millones de dispositivos del Internet Of Things (IOT) enviando un registro cada 10 segundos o 1000 servidores generando 3.000 registros por segundo cada uno. Toda esta información debe ser analizada y almacenada para poder ser visualizada y usada posteriormente de la forma más útil posible. Para lograr este ambicioso objetivo, es necesario dividir el trabajo del análisis y la representación entre diferentes equipos y construir un novedoso sistema de monitorización de logs distribuido. *Loginson* fue creado para solventar esta necesidad y partió de los siguientes cuatro pilares:

1. **Centralización de logs**: En grandes centros de datos el sistema de log por defecto es Syslog [97], tanto para eventos de equipamiento de red como del resto de los sistemas físicos o virtualizados. No obstante, configurar todos estos equipos de forma que sus registros sean enviados a diferentes nodos de forma equilibrada es un problema complejo y que requeriría una gran intervención manual. Por este motivo, que todos los equipos envíen siempre a un conjunto de colectores fijo y sean estos los encargados de realizar el reparto final de los logs es la solución mas sencilla y funcional. Esto implica por tanto la presencia de balanceadores de carga dentro de *Loginson*.

2. **Preprocesado de registros**: Tal y como se ha explicado anteriormente, el preprocesado de los registros es la componente más compleja y dependiente del formato de los registros. Por este motivo, es necesario construir diferentes filtros para cada tipo de registro, lo cual debe ser a su vez una tarea flexible para hacer el sistema escalable. Al ser el preprocesado de registros un componente sin restricciones temporales, el balanceador citado anteriormente puede repartir entre tantas instancias de preprocesadores cómo sea necesario.

3. **Simplificación del almacenamiento**: Una vez los logs han sido preprocesados, deben ser almacenados para su posterior consulta. Estos datos, tienen cierta localidad temporal, por lo que normalmente los datos mas consultados serán también los más recientes y por ende, deben residir un tiempo en memoria para agilizar las consultas. A su vez, el tipo de almacenamiento utilizado debe ser simple y rápido de consultar, ya que las consultas típicamente serán mas simples que las de una base de datos tradicional es posible prescindir de ciertas funcionalidades a favor de velocidad.

4. **Representación gráfica**: Finalmente, es necesario convertir los datos almacenados previamente en una visualización amigable para el usuario. Para ello es posible recurrir a software abierto como *Grafana*[1] o *Kibana*[2].

3.1.1. Estado del arte

La mayor parte de los sistemas big data distribuidos enfocan su desarrollo a una escalabilidad horizontal lo más óptima posible a la vez de intentan proporcionar una cierta facilidad de uso y mantenibilidad. Estas decisiones de diseño, que en un principio son deseables en cualquier software, causan en muchas ocasiones una infrautilización el hardware disponible. Los autores de [98] atribuyen este problema a que estos sistemas se centran en solucionar problemas del tipo *Big Data*. El escenario más común en este mundo suele partir de un conjunto de equipos que ya contienen un conjunto muy grande de datos almacenado previamente y cuya volumen de ingesta de nueva información es relativamente baja en comparación a la información previamente almacenada. Para poner en contexto la ineficiencia de estos sistemas, en [99] llegan a la cifra récord de un millón de escrituras por segundo en una base de datos *Cassandra*[3] recurriendo a más de 300 máquinas virtuales.

Balanceo de carga

Uno de los métodos más sencillos para construir un sistema distribuido es el balanceamiento de carga de las peticiones, o en el caso de Loginson de la ingesta de logs. Existen muchas opciones actualmente para hacer este tipo de procesos de forma genérica, por ejemplo *Apache Kafka* o *Apache Storm*. Este último además permite generar y construir pipelines dinámicamente que además de gestionar la ingesta pueden realizar cualquier tipo de preprocesado de los datos antes de su almacenamiento o representación gráfica a tiempo real. Los dos sistemas mencionados son ampliamente estudiados en la Subsección 3.2.4.

Apache Flume[4] es una alternativa a Kafka orientada más específicamente al procesamiento de Logs. *Flume* envía de forma activa los datos recibidos y de acuerdo a unos filtros predefinidos al grupo de destinos que haya sido definido en la configuración. Algunos de los posibles destinos puede ser HDFS [100], Elasticsearch,

[1] `http://grafana.org`
[2] `https://www.elastic.co/products/kibana`
[3] `http://cassandra.apache.org/`
[4] `https://flume.apache.org/`

entre otros muchos. En [94] se hizo una prueba de rendimiento de *Flume* bajo distintas configuraciones para obtener sus rendimientos límite. En este estudio se probó en el escenario más beneficioso posible con todos los resultados escritos en memoria RAM, llegando a alcanzar el millón y medio de logs por segundo al utilizar todos los cores disponibles. Dado que el objetivo de Loginson pretende abarcar hasta 3 millones de logs, y el sobrecoste de reenviar los flujos por una o varias interfaces no se había tenido en cuenta, se descartó *Flume* como un balanceador suficientemente rápido.

Logstash[5] es probablemente uno de los sistemas más populares en cuanto a gestión de Logs se refiere. *Logstash* permite realizar multiples manipulaciones de los logs, así como filtrados complejos, transformaciones y copias de los mismos a distintos sistemas de almacenamiento. No obstante, su flexibilidad le ha pasado factura haciendo a *Logstash* incapaz de procesar mas de unas pocas decenas de miles de eventos por core en los mejores escenarios [94].

FluentD[6] es una alternativa a *Logstash* que presume de haber alcanzado tasas de hasta 800,000 eventos por segundo [101] en sus sistemas en la nube aunque sin ningún detalle del hardware subyacente [102]. El único dato actual es el ofrecido en su web dando una tasa de hasta 13 mil eventos por segundo y núcleo de CPU [103].

Sistemas de almacenamiento y Bases de Datos

Una vez se han recibido los logs en el sistema, el balanceador de carga debe distribuirlo a distintos nodos de almacenamiento que se encargarán de indexar y filtrar la información para su posterior representación gráfica. Existen muchas opciones para almacenar estos datos, entre ellos las opciones clásicas orientadas de Big Data como HDFS [100] o la previamente mencionada base de datos *Cassandra*. Sin embargo es necesario poder lograr búsquedas rápidas así como la capacidad de un filtrado, mientras se mantiene el rendimiento de al menos 3 millones de eventos por segundo. Por otro lado, están las bases de datos estructuradas basadas en el funcionamiento *clave-valor*, las cuales no encajan del todo con la filosofía de estructura flexible que tienen los Logs en su naturaleza.

ScyllaDB[7] nació como una alternativa a *Cassandra*, reescribiendo su código en un lenguaje de más bajo nivel (C/C++), hasta el punto de reescribir toda la pila

[5] `https://www.elastic.co/products/logstash`
[6] `http://www.fluentd.org/`
[7] `http://www.scylladb.com/`

de red, intentando de esta forma obtener el máximo rendimiento por nodo manteniendo su flexibilidad y escalado horizontal. Según algunos resultados aportados en la web de *ScyllaDB* [104], son capaces de alcanzar ampliamente 100.000 operaciones por minuto (es decir, de lectura y escritura combinadas) por cada equipo físico, si este tiene una capacidad de cómputo muy elevada. Aunque en comparación con *Cassandra* requeriría aproximadamente 10 nodos según dichos tests para llegar al mismo número de transacciones por segundo, siguen siendo números muy inferiores al objetivo de Loginson.

Existen otras soluciones en el mercado, como *MapRDB*[8] que promete una ingesta de hasta 100 millones de "puntos" por segundo y con replicación 3 en 4 nodos gracias a su sistema de ficheros a medida. *Splunk* es una alternativa de pago a *Elasticsearch* con un rendimiento de unas 80.000 inserciones por segundo y nodo. No obstante, al ser opciones comerciales fueron excluidas de nuestros experimentos por no que no pudimos evaluar si realmente *MapRDB* era o no una opción válida como almacén de Logs y búsqueda de Loginson.

Preprocesado de logs y visualización

Loginson cómo sistema de monitorización de logs necesita procesar, filtrar, serializar y graficar los resultados. El estado del arte de dicha funcionalidad no corresponde al objetivo de la sección de este capítulo y se encuentra distribuido entre la Subsección 3.2.2 y el Capítulo 4.

3.1.2. Arquitectura

Tras analizar las diferentes opciones para el procesamiento de logs se pueden deducir mayoritariamente dos hechos: (I) No hay ningún sistema capaz de preprocesar en un único equipo un volumen de varios millones de logs por segundo de forma efectiva, y (II) los sistemas capaces de gestionar un volumen de datos de esas dimensiones son distribuidos y orientados a un escalado mayoritariamente horizontal, con graves deficiencias en el escalado vertical. El escalado horizontal permite de forma sencilla –añadiendo más equipos– solventar un problema complejo de cualquier dimensión. No obstante, el coste de un número elevado de equipos puede ser demasiado elevado, sobretodo en comparación con el sistema que deseamos monitorizar. Por este motivo es necesario optimizar los sistemas actuales y diseñar una arquitectura capaz de explotar en mayor medida el escalado

[8] https://www.mapr.com/

vertical, pudiendo por tanto reducir el número de equipos necesario y a su vez el coste global del sistema. Con este objetivo en mente y de acuerdo a los pilares de Loginson, se diseñó una arquitectura en formato pipeline, el cual se puede desglosar en 4 etapas fundamentales: (I) Recepción y distribución de logs, (II) Almacenamiento, (III) filtrado y preprocesamiento y por último (IV) Visuzalización.

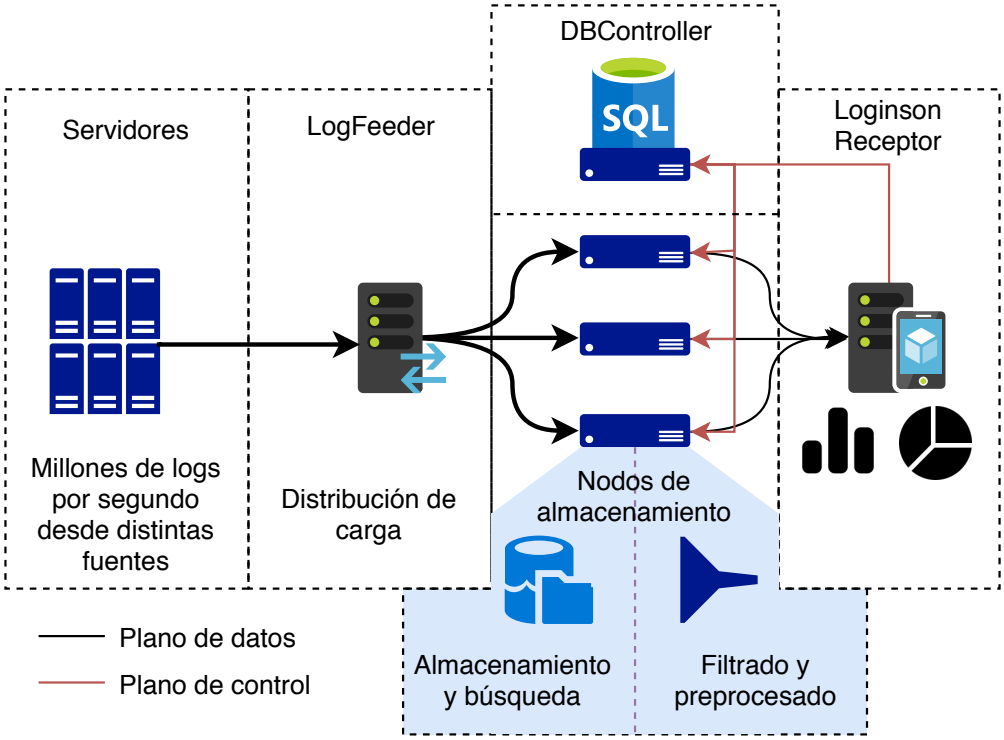


Figura 3.1. Arquitectura de Loginson

Recepción y redistribución de logs

La gestión de un datacenter tiende a presentar problemas, los cuales se acentúan al hablar de varios datacenters distribuidos geográficamente. Es común que distintos servicios sean gestionados por distintos grupos, lo que a su vez complica y ralentiza operaciones de mantenimiento que afecten a todo el datacenter (o conjunto de ellos). Por este motivo, es necesario la introducción de una primera etapa de recepción y redistribución de logs (la cual llamaremos *LogFeeder*). Al introducir un único punto de recogida de logs, –o en su defecto un subconjunto pequeño de ellos–, es posible realizar un balanceo de la carga entre distintos nodos de almacenamiento sin necesidad de reconfigurar manualmente los distintos

servicios y equipos, y por ende, reduciendo los costes y problemas de mantenibilidad. Esta etapa necesita procesar un gran número de eventos por segundo, lo que impide que se puedan hacer operaciones complejas como análisis o filtrado de los mismos. Estas operaciones deben ser delegadas a etapas posteriores. No obstante, la etapa de recepción puede añadir cierto valor adicional realizando operaciones sencillas, como el marcado temporal de los logs de entrada. Esta funcionalidad puede ser crucial en muchos entornos, ya que al haber miles o incluso millones de equipos enviando logs, es altamente probable que se pierda la sincronización horaria entre algunos de estos equipos debido a múltiples de causas (Problema de comunicación con el servidor NTP, problema hardware, fallos de configuración, etc.). La desincronización entre eventos y equipos podría potencialmente causar que un problema en uno de ellos no pudiese ser relacionado con un evento en otro equipo.

La función de esta primera etapa es conceptualmente sencilla, capturar a la mayor velocidad posible los logs de entrada evitando perder eventos y reenviarlos de acuerdo a un algoritmo de balanceo de carga (en el diseño original Round-Robin) a un conjunto de nodos de almacenamiento.

Almacenamiento

Dada la naturaleza de los logs, dentro de una determinada categoría o fuente es normal que los eventos sean accedidos en bloques en un cierto intervalo de tiempo. Una base de datos tradicional SQL ofrece mucha funcionalidad que no es necesaria para el problema de almacenamiento de logs. Además, están orientadas a almacenar datos estructurados y es común que los logs no lo sean. De acuerdo al estado del arte, este tipo de bases de datos, al igual que las distribuidas, tienen un sobrecoste elevado en búsqueda y escritura que no soportaría la tasa de varios millones de escrituras por segundo y aun menos búsquedas de esos volúmenes. Por este motivo se decidió diseñar un sistema de almacenamiento propio.

Tal y como se muestra en la Figura 3.1, el sistema de almacenamiento se divide en dos tipos de nodos: (I) el nodo de gestión (*DBController*) y (II) los nodos de almacenamiento. El nodo gestión almacena algunos metadatos en una base de datos tipo SQL (fecha de creación, procedencia, tipo, etc.) acerca de los logs y el nodo físico en el que se encuentran almacenados. Gracias a este diseño las búsquedas se realizan de forma similar a como funciona el modelo *map-reduce*, ya que el nodo de gestión realiza una búsqueda previa para localizar los ficheros que contienen

la información deseada y son los nodos que contienen dichos logs los que realizan una búsqueda fina.

Los datos retransmitidos en la etapa anterior (*LogFeeder*) son recibidos y almacenados en buffers intermedios. Al llenarse dichos buffers, se escribe en disco su contenido en un único bloque optimizando así los procesos de escritura.

Preprocesamiento, filtrado y visualización

Para poder visualizar los datos es necesario realizar un preprocesado y estructuración de los mismos. Este proceso se realiza en los nodos de almacenamiento para ahorrar transferencias de datos innecesarias. Tanto el preprocesado como el filtrado se realiza tanto a tiempo real como bajo demanda. Loginson en su desarrollo inicial utiliza *Kibana* y *Elasticsearch* como sistemas de representación de datos. Esto implica que el sistema de almacenamiento debe serializar los logs en formato *JSON* para poder funcionar con dichos sistemas. A su vez, dichos sistemas presentan un cuello de botella importante en términos de volumen de ingesta de datos así cómo de representación de los mismos. Por este motivo, la representación de datos en tiempo real requiere del uso de agregaciones o en su defecto de realizar un muestreo sobre los registros almacenados. Aunque el muestreo o la agregación de los datos es una aproximación válida para ofrecer un idea general del estado del sistema en un uso diario, puede no ser una aproximación válida en caso de que exista una incidencia. Por este motivo Loginson incorpora el componente final de la cadena *Loginson Receptor* el cual pretende dar solución al uso diario y al uso exhaustivo en caso de incidencia.

Las funciones de este componente son las siguientes: (I) recibir los datos de los nodos de almacenamiento e insertarlos en la BBDD utilizada para la representación gráfica (*Elasticsearch*), y (II) proveer a través de la interfaz gráfica (*Kibana*) una funcionalidad capaz de hacer "zoom" sobre los datos mostrados solicitando a los sistemas de almacenamiento los datos que desea el usuario.

3.1.3. Evolución hacia el procesamiento de paquetes

En esta sección se ha discutido el estado del arte sobre los sistemas de monitorización a nivel de aplicación basados en Logs. Estos sistemas son funcionales en muchos aspectos y ampliamente utilizados por la industria, no obstante, tienden a presentar un coste elevado que los hace inaccesibles para muchas empresas. Cómo solución se propuso Loginson, un sistema que partía de utilizar la mayoría de las

ideas de los sistemas actuales reorientando su implementación hacia un enfoque con un escalado más vertical. La arquitectura en formato de pipeline así como su etapa concordaba mucho con un sistema clásico de procesamiento en streaming, no obstante, al igual que los sistemas de logs y bases de datos estos sistemas de procesamiento de streaming estaban demasiado enfocados en el escalado horizontal proporcionando un rendimiento mono-equipo muy inferior a las capacidades del hardware. Buscando dar el siguiente paso, se planteó la idea de formar un sistema de streaming, con una capacidad que pudiese permitir lo que no se había visto hasta entonces: el análisis de paquetes distribuido en tiempo real. Con esto en mente y con muchas lecciones aprendidas de Loginson nació Wormhole.

3.2. Wormhole

La mayoría de las herramientas de streaming se definen como soluciones de alto rendimiento, capaces de trabajar a alta velocidad. Posiblemente esto sea verdad para los típicos problemas de Big Data, donde incluso a una relativa "alta velocidad", los datos se distribuyen fácilmente, y normalmente, la complejidad está en el procesado de los datos, lo cual requiere una gran cantidad de equipos. Sin embargo, la captura de tráfico en redes de 40 o 100 Gbit/s representa una diferencia enorme en requisitos de ingestión y comunicación con respecto a los clásicos problemas de Big Data que requieren motores de streaming.

Con la adopción de soluciones Big Data para la gestión y monitorización de redes, se requieren nuevas soluciones de alto rendimiento para las etapas de ingestión y comunicación, porque no tiene sentido construir un clúster para monitorizar una red. Por ello, se propone un nuevo motor de streaming, llamado Wormhole, que aprovecha las últimas tecnologías de red, y permite un crecimiento tanto vertical como horizontal con el objetivo de funcionar a alta velocidad sin incrementar el número de equipo radicalmente.

Usando Wormhole como herramienta para ingestión y distribución de datos, junto con otras soluciones del ecosistema Big Data, se está en disposición de diseñar sistemas completos capaces de crecer y adaptarse al tamaño de la red a monitorizar, proveyendo capacidad completa de análisis, desde estadísticas en tiempo real a predicciones con Machine Learning, a partir de los datos capturados.

3.2.1. Motivación

La gestión de red es un reto que no tiene fin, ya que su dificultad crece cada año, a medida que evolucionan las redes. Esta transformación de las redes afecta directamente a los procesos de monitorización clásicos, que se basan en equipos de monitorización muy específicos y muy caros. Es importante destacar que la monitorización de red juega un papel muy importante en la gestión de la red. Por ejemplo, en los centros de datos actuales, la monitorización de red permite planificar el uso de equipamiento de red a partir de las necesidad de uso reales, validando la interacción entre los diferentes componentes, e incluso detectando anomalías entre ellos o haciendo análisis forense de las intrusiones, entre otras tareas.

Cuando una conexión de red se monitoriza de forma pasiva, todo el tráfico que la atraviesa se captura, de modo que el reto es proporcional a la velocidad de la conexión. Este problema es recurrente ya que la velocidad de las conexiones se incrementa cada cierto tiempo, y desafortunadamente el hardware de monitorización no tiene esa misma capacidad de evolución en términos de CPU, memoria y almacenamiento. Una posible solución es no monitorizar todo el tráfico que atraviesa la conexión, sino que se muestrea. Por tanto, algunas de las soluciones actuales de monitorización son "a medida" y no proveen una visión completa de la red, o esa visión está resumida excesivamente, o incompleta.

Una de las primeras propuestas de uso de plataformas Big Data para la monitorización de red es [13]. Sin embargo, aunque la mayoría de herramientas de procesamiento (Hadoop, Spark, Hive, etc.) pueden ser usadas para procesar datos de red, al fin y al cabo, son datos en la parte de ingestión de datos a alta velocidad y su distribución entre equipos no es tan sencillo. La mayoría de herramientas de captura y comunicación de datos que afirman presentar un gran rendimiento, lo obtienen si se emplea un gran número de equipos. Sin embargo, para monitorizar una red es obligatorio capturar los datos en cada punto de monitorización o captura a tasa de línea. Además, resulta imposible procesar estos datos a tasa de línea usando las herramientas tradicionales de Big Data, por lo que es necesario un mecanismo de comunicación de alta velocidad entre equipos de procesamiento para repartir los datos. Con objetivo de resolver estas limitaciones, Wormhole provee capacidades de captura a alta velocidad y permite comunicar y repartir los datos capturados a tasa de línea en los diferentes puntos de la red, entre los diferentes equipos.

3.2.2. Estado del arte

Monitorización en único equipo

Las soluciones clásicas de monitorización consisten en un único equipo, y se han mejorado con el paso de los años gracias a los avances que han ocurrido en las capacidades de captura [12], en técnicas inteligentes de reducción del ancho de banda como el muestreo de paquetes o NetFlow/IPFIX [25], y el uso de soluciones hardware puras [105], permitiendo al equipamiento de red (routers y switches) crear resúmenes del tráfico que manejan internamente. Sin embargo, este resumen requiere una potencia de computo alta, y tanta memoria como número de flujos concurrentes se quiere monitorizar. Este número de flujos es proporcional a la velocidad de la conexión donde se conecta el equipo. Debido a estas necesidades computaciones, el equipamiento de red normalmente muestrea los paquetes [29], un método que permite obtener una vista general de la red, descartando paquetes siguiendo alguna regla, generalmente se eligen de 1 a N paquetes de manera aleatoria.

La monitorización de paquetes se puede usar también en equipamiento específico para captura de tráfico. Sin embargo, este método no es aceptable en la mayoría de los casos porque perder paquetes significa también perder flujos [30], y esta situación puede llevar a no detectar ataques lentos y continuados, como puede ser un slow-speed DDoS [31]. Por tanto, es importante alcanzar la captura de paquetes con cero pérdidas. En [106] se presenta la primera solución de monitorización a 100 Gbit/s. Sin embargo, esta solución, aunque efectiva, es un poco cara en cuanto a recursos, ya que se requieren 10 máquinas para captura y análisis.

Los métodos de análisis de red distribuidos se han descartado típicamente porque, aunque dividir el tráfico de red es un proceso sencillo, agregarlo de nuevo para su análisis es una tarea compleja. Es por ello que el mundo académico y la industria han hecho un gran esfuerzo en el desarrollo de motores de captura y análisis muy eficientes [12] que se pueden ejecutar en un único equipo. Sin embargo, es muy común en empresas de gran tamaño que el tráfico de red esté dividido, esto es, una empresa necesita analizar su red, independientemente de si solo tiene un centro de datos o decenas de ellos distribuidos a lo largo de un país, un continente o alrededor del mundo. Esto hace que la compañía tengo muchas soluciones independientes que no siempre le permiten obtener una visión común de toda la red.

Motores de streaming actuales

Si el crecimiento vertical de los procesos de monitorización es un problema, y el procesamiento distribuido también lo es, el uso de sistemas Big Data, que precisamente se crearon para tratar este tipo de problemas, podría ser una buena solución.

Actualmente, existen varias soluciones que permiten distribuir el procesamiento entre varios equipos, típicamente siguiendo lo que se conoce como arquitectura Lambda [107]. La arquitectura Lambda tiene dos elementos principales: una parte "tiempo real" (o streaming), que se encarga del procesamiento de los datos según se generan (u obtienen); una parte "off-line" (o batch), que ejecuta procesamientos en bloques de datos previamente almacenados. Procesar paquetes de red en "batches" es una solución ya aportada por Lee y Lee en [108], donde también se identifica como una solución bastante ineficiente, ya que requiere un alto número de equipos y, además, no resuelve el problema de almacenar o capturar los datos a alta velocidad. Incluso aunque los datos de semanas o meses pasados son importante e interesantes de analizar, es esencial procesar los datos en cuanto se dispone de ellos cuando ocurren un incidente. Este es el motivo por el que también interesa analizar los motores de procesamiento en streaming.

Existen varias herramientas Big Data para procesamiento en streaming, cada una de ellas con características únicas. Sin embargo, suelen compartir algunas de las características, como puede ser (i) diseño orientado a escalar horizontalmente; (ii) minimizar la latencia entre mensajes; (iii) garantizar que el mensaje llega a los clientes. Algunas de estas herramientas, como Spark Streaming [109], agrupan los mensajes en bloques, intentando obtener un mejor rendimiento a costa de sacrificar latencia. Otras, como ZeroMQ [110], intentan obtener la mayor eficiencia y la latencia más baja usando lenguajes de programación de alto rendimiento (C++). Apache Kafka [111], sin embargo, crea un log de cada mensaje enviado y siempre almacena los mensajes que recibe para garantizar la entrega y hacer seguimiento de los datos.

Existen varias publicaciones previas que han intentado analizar el impacto del diseño en el rendimiento para la mayoría de estos motores de Streaming [112–114]. Sin embargo, estos trabajos no se han enfocado en evaluar o comparar el rendimiento a nivel de red (comunicaciones), sino otros aspectos como pueden ser las capacidades de los motores para adaptarse, garantizar el reparto o entrega de los mensajes, etc. Por ejemplo, en [112] se comparan tres de las más impor-

tantes soluciones de streaming: Storm [115], Flink [116] y Spark Streaming. Sin embargo, en todas las pruebas se hace uso de Kafka para enviar datos y Redis[9] para almacenar los resultados, y aunque esta aproximación pueda ser muy común. no permite evaluar el rendimiento máximo de cada una de las soluciones. Es más, la tasa máxima de recepción de estos sistemas es de 180 000 eventos por segundo, usando 40 equipos. Estas tasas están muy alejadas de las necesidades reales de una conexión de red ideal de 100 Gbit/s, donde, en el caso peor con paquetes de 60 bytes, se alcanzan 148.8 millones de paquetes por segundo.

Otro trabajo, presentado en [113], es similar al previo, excepto que alcanza 20 millones de eventos por segundo con un número menor de equipos (7) usando Storm, lo cual demuestra una importante discrepancia si se compara con [112]. Los mismos tres motores de streaming se comparan de nuevo en [114], junto con Hadoop en modo batch, y usando Kafka para los datos de entrada. Este trabajo se enfoca en el análisis de los recursos de las diferentes soluciones (CPU y RAM), en vez del ancho de banda y la latencia. Por tanto, aunque interesante, no es una comparación que nos ofrezca lo que buscamos. Además, se accede a disco para obtener los datos para las pruebas, en vez de recibir los datos directamente de otro equipo, por lo que el ancho de banda obtenido será completamente diferente de los trabajos anteriores. En otro trabajo [117], se centran en el análisis de uno de los motores de streaming en detalle, Kafka, usando una configuración de bajo coste, lo cual no cumple con nuestros requerimientos de rendimiento o análisis realista con hardware actual.

Dado que no existe en el estado del arte ningún trabajo que cumpla con nuestras necesidades, además de las discrepancias entre resultados de los diferentes trabajos, y la falta de resultados que muestren rendimientos razonables para resolver las necesidades de la monitorización, lo razonable es realizar nuestros propios experimentos para comparar el rendimiento de los motores de streaming más comunes. En nuestras pruebas hemos tenido que descartar Apache Flink debido a la limitación que parece tener para determinar donde se ubican las tareas[10].

3.2.3. Evaluación de los motores de streaming actuales

Una de las mayores reticencias para usar monitorización y análisis distribuido es el número de equipos que son necesarios para llevar a cabo el análisis, y por

[9] `https://redis.io/`

[10] `https://stackoverflow.com/questions/51810533`

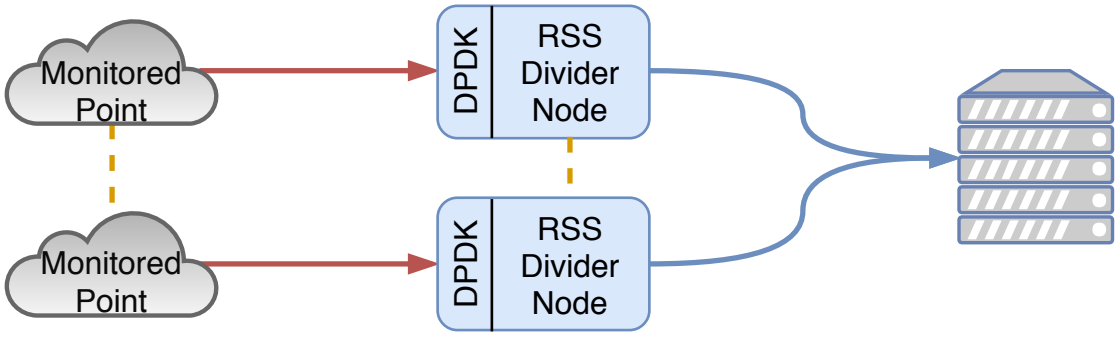


Figura 3.2. Monitorización de red basada en la captura de tráfico con DPDK y distribución usando colas RSS.

tanto, el coste final del sistema. Si usamos una solución que escale muy bien horizontalmente, pero desaproveche una gran cantidad de recursos de los equipos, será necesario un número considerable de equipos para procesar el tráfico de red a la velocidad de 100 Gbit/s o superior. Por tanto, no tiene sentido usar un pequeño centro de datos para monitorizar otros centros de datos.

Para evaluar si las soluciones de streaming actual puede resolver el problema de la monitorización distribuida y el análisis, se define un caso de prueba, que se muestra en Figura 3.2. Dado uno o más puntos de observación de la red a 100 Gbit/s, un equipo usando DPDK captura todos los paquetes de la red y realizar una pre-distribución en los diferentes cores del equipo usando las colas RSS, porque se ha demostrado que es una muy buena solución para el balanceo de tráfico de red real [39]. Un proceso, usando una de las actuales soluciones de Streaming, recoge los paquetes de las colas RSS, y los envía a los equipos de procesamiento, que se encargarán del análisis. Gracias a las colas RSS, todos los paquetes de la misma conexión (flujo) se envían al mismo equipo de cómputo, por lo que es posible almacenarlos y extraer la información importante a muy bajo nivel. Una vez se ha propuesto el escenario de prueba, se va a evaluar cuantos equipos y CPUs requiere cada una de las soluciones de streaming mejor valoradas por su rendimiento: (i) ZeroMQ 4.2.2; (ii) Apache Spark 2.3.1; (iii) Apache Storm 1.2.2; y (iv) Apache Kafka 2.11.

Evaluar estas soluciones es un proceso complejo, porque cada una de ellas tiene diferentes paradigmas de programación y comunicación, que generalmente no son similares, así como una configuración muy amplia. Las diferentes combinaciones pueden llevar a cometer errores fácilmente, afectando a la comparación. Por tanto, para una primera evaluación se decide realizar un test lo más simple posi-

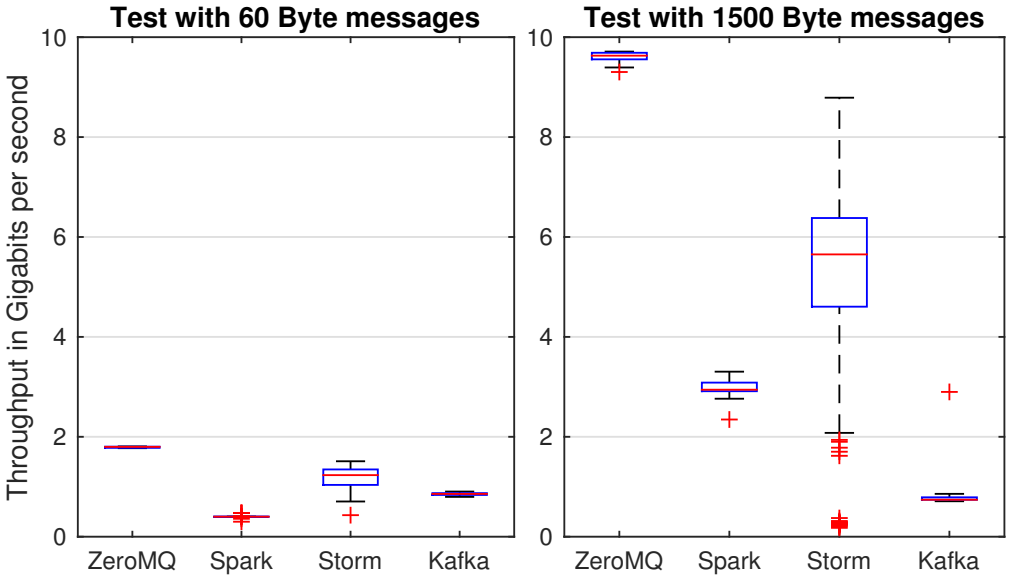


Figura 3.3. Resultados de los test para cada uno de los motores de streaming. Mensajes de 60 bytes (izquierda) y 1500 bytes (derecha). Cuanto más alto, mejor.

ble para evaluar el ancho de banda y la latencia, usando valores de configuración por defecto. Para implementar el escenario de test, se usan dos equipos que disponen de una tarjeta de red Ethernet Mellanox Connect-X 5 100 Gbit/s. El primer equipo tiene dos *Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz* con 32GB de RAM, mientras que el segundo tiene dos *Intel Xeon Gold 6126 @2.60GHz* con 48GB de RAM. El sistema operativo es Gentoo con Linux 4.14.7 y Oracle Java 8. Para el test de ancho de banda, se usan dos procesos, uno en cada equipo. Un proceso envía al otro mensajes de tamaño fijo, 60 bytes o 1500 bytes, para emular el peor y el mejor caso. Después de enviar un millo de paquetes, se guarda el valor medio del ancho de banda obtenido, y el experimento se repite 10000 veces. Cuando es posible, la solución de streaming se configura con las características de retransmisión y ACK deshabilitadas.

En Figura 3.3, se puede observar como ZeroMQ y Apache Storm estan por encima del resto. Esto se debe a que Apache Storm usa colas de mensajes similares a ZeroMQ. Sin embargo, debido al uso de la Java VM, la media y varianza de Storm son peores que ZeroMQ. Los resultados de Spark y Kafka son peores, siendo Kafka el peor de todos. Los resultados Apache Kafka results son tan malos porque almacena en disco todos los mensajes recibidos antes de servirlos. En las pruebas con Kafka se ha usado un sistema de ficheros en memoria para reducir el impacto,

Tabla 3.1: Comparación de la media (μ) y desviación estándar (σ) del ancho de banda en Gbit/s para una conexión de 100 Gbit/s con solo dos threads. Tamaño de mensajes de 60 y 1500 bytes.

Tam. Msg.	Rendimiento	Wormhole	ZeroMQ	Spark	Storm	Kafka
60	μ	9.35	1.79	0.40	1.19	0.85
	σ	0.51	0.01	0.03	0.19	0.03
1500	μ	23.94	9.61	2.98	5.19	0.94
	σ	0.04	0.01	0.02	3.72	0.38

pero aún el uso del disco genera un sobrecoste que hace a esta solución la peor de todos en cuanto a rendimiento por thread.

Aunque la prueba de ancho de banda muestra que ZeroMQ es capaz de alcanzar los 9,6 Gbit/s para un tamaño de mensaje de 1500 bytes, solo alcanza los 2 Gbit/s cuando el tamaño de mensaje es de 60 bytes. Si se asume un comportamiento del ancho de banda lineal, para monitorizar a 100 Gbit/s se necesitaría entre 10 y 50 threads únicamente dedicados a comunicaciones, tanto el receptor de tráfico como en los nodos de procesamiento. Por tanto, se necesitan un gran número de equipos para implementar esta solución, y probablemente su rendimiento sea peor que el presentado en [106], el cual funciona con cualquier tamaño de mensaje con 10 equipos.

La latencia es otra de las métricas a tener en cuenta en los sistemas de monitorización, pero no es tan crítica como el ancho de banda. Nos permite tener un sistema reactivo automatizado cuando se produzca un evento en la red. Para medir la latencia, se usa el mismo despliegue experimental que para el ancho de banda, y se mide el RTT con tres threads diferentes: (i) Trasmisión y marcado temporal del mensaje; (ii) reenvío del mensaje; y (iii) recepción y medida final. Para disponer de una medida precisa, los tres threads se simplifican al máximo y se asegura que los threads (i) y (iii) siempre se ejecuten en el mismo equipo para evitar problemas de sincronización del reloj. En la prueba se envían 100.000 mensajes, de tamaño 60 a 1500 bytes, marcados temporalmente con precisión de nanosegundos usando la misma técnica que en [18].

Los resultados de retardo obtenidos se muestran en Figura 3.4. Son bastante similares a los obtenidos para el ancho de banda. ZeroMQ destaca de nuevo, mientras que Spark y Storm sufren debido al rendimiento de la Java VM. Los resultados de Apache Kafka son de nuevo bastante malos por la misma razón que los obtenidos en las pruebas de ancho de banda.

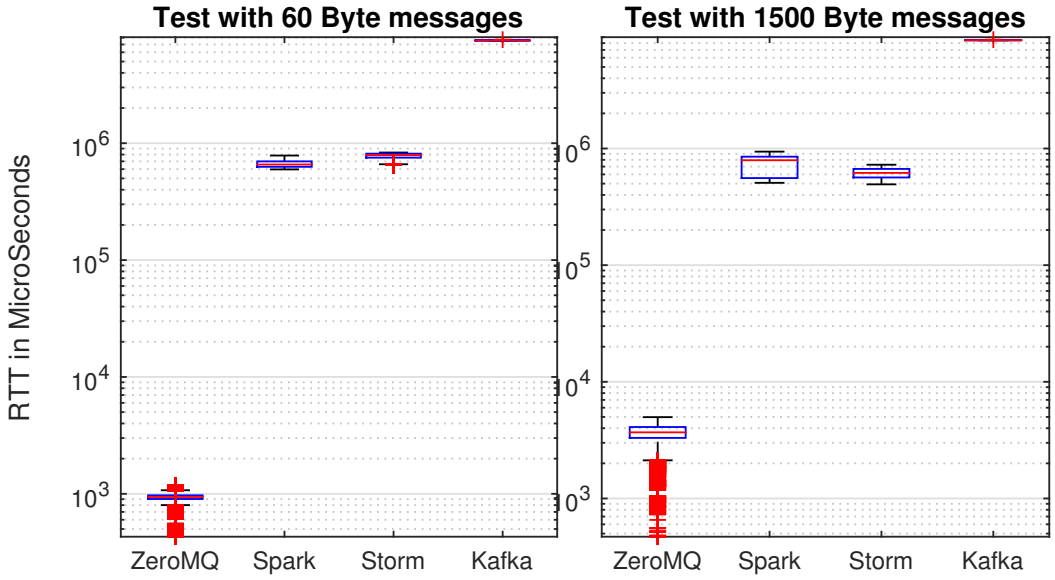


Figura 3.4. Test de RTT para cada uno de los motores de streaming. Tamaño de mensaje de 60 bytes (izquierda) y 1500 bytes (derecha). Escala logarítmica. Más bajo, mejor.

Tabla 3.2: Comparación de la media (μ) y desviación estándar (σ) de la latencia en ms para una conexión de 100 Gbit/s con solo tres threads. Tamaño de mensajes de 60 y 1500 bytes.

Tam. Msg.	RTT	Wormhole	ZeroMQ	Spark	Storm	Kafka
60	μ	3.29	0.94	673.59	777.12	7576.43
	σ	0.40	0.05	57.56	46.12	76.29
1500	μ	4.48	3.68	727.58	615.39	8473.81
	σ	1.15	0.50	147.44	63.58	85.09

3.2.4. Diseño de Wormhole

Una vez obtenidos los resultados, se decide qué características de los motores evaluados son útiles para lo que podríamos considerar un motor de Streaming orientado a monitorización de red:

1. **Implementación usando lenguajes de alto rendimiento**, para evitar por ejemplo las ineficiencias de lenguajes como Java [118].

2. **Uso eficiente de threads**, de modo que un único thread debería ser capaz de recibir datos a 10 Gbit/s en el peor de los casos (paquetes de 60 bytes).

3. **Simple:** no implementar funcionalidades innecesarias o que supongan una sobrecarga, como puede ser ACKing.

4. **Usar batches**: permiten mejorar el ancho de banda, a costa de un pequeño incremento de la latencia.

5. **Diseño escalable** de modo que sea eficiente el escalada horizontal y vertical.

6. **Fácil de usar:** incluir herramientas para auto-despliegue, control, balanceo, etc.

Por lo tanto, la implementación de Wormhole se ha realizado en lenguaje C, y los mensajes se envían en batches. Además, se provee un API simple que facilita al usuario definir los procesos de streaming. Finalmente, el diseño escala muy tanto vertical como horizontalmente. De este modo, se obtiene máximo rendimiento posible, al tiempo que se maximiza la compatibilidad con un despliegue ya existente. Esto significa que el sistema usa sockets BSD para el envío de mensajes, y se evitan el uso de elementos SDN (como el usado en [119]) o implementaciones con DPDK a medida.

Con el objetivo de hacer que la arquitectura de Wormhole sea similar a otros motores de streaming, se han definido los siguientes elementos:

1. **Holes**: Un *hole* es un thread o proceso que usa la librería *Wormhole*, porque cada mensaje se asocia a un *hole*, donde se procesa. Un equipo puede ejecutar un número arbitrario de *holes*, independientemente de sus conexiones. Sería equivalente a un *Bolt* y un *Spout* en terminología de Storm [115].

2. **Worm**: Es el término usado para cada mensaje, porque "viajan entre diferentes *holes* del *Universe*".

3. **Universe**: Es el término usado para definir la topología del *hole*, y como los *holes* interactúan entre ellos. Sería similar a *Topology* en terminología Storm.

4. **Zeus**: Es el encargado de coordinar y desplegar los *holes* definidos en el *Universe* entre los diferentes equipos. Equivale al *Nimbus* en Storm.

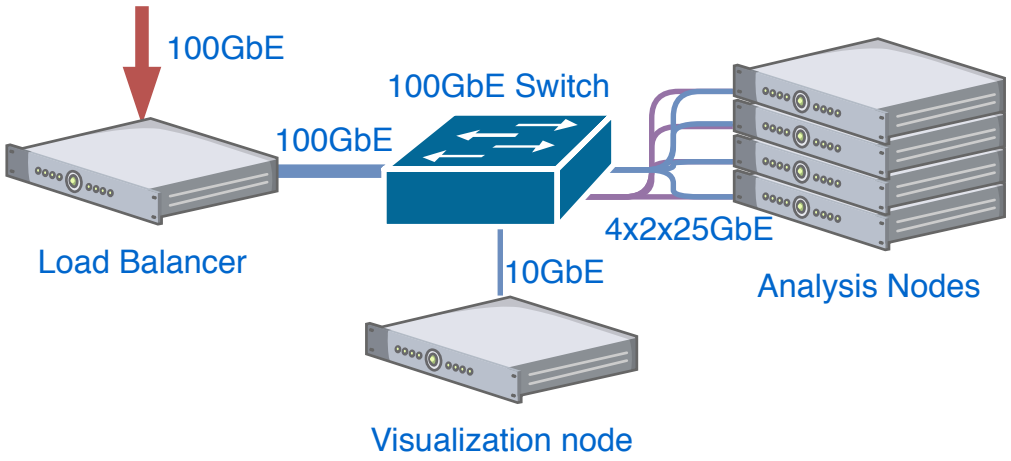


Figura 3.5. Banco de pruebas para la evaluación de Wormhole a 100 Gbit/s.

Rendimiento de Wormhole

Los resultados de las pruebas de ancho de banda y latencia para evaluar el rendimiento de Wormhole se muestran en las tablas 3.1 y 3.2. Aprovechando eficientemente los sockets de Linux, se consigue un ancho de banda de 24 Gbit/s para tamaño de mensaje de 1500 bytes, y 10 Gbit/s para tamaño de mensaje de 60 bytes. Sin embargo, dado que la latencia no es una prioridad, Wormhole es en esta prueba un orden de magnitud más lento que ZeroMQ para mensajes pequeños (60 bytes). Pero presenta mejores resultados que los otros tres motores analizados, y resultados similares a ZeroMQ con mensajes grandes (1500 bytes).

A partir de los resultados obtenidos, se ha diseñado un banco de pruebas para procesar datos de red a 100 Gbit/s, que se muestra en Figura 3.5. La arquitectura del banco de pruebas consiste en equipo que realizar el balanceo de la carga (load balancer node), cuatro equipos para análisis (analysis nodes) y un equipo para visualización (visualization node). Todos ellos ejecutan un sistema operativo *Gentoo* con *Kernel Linux 4.14.7*. El equipo balanceador de carga dispone de dos procesadores *Intel Xeon Gold 6126 @2.60GHz* y dos tarjetas *Mellanox Connect-X 5*, una de ellas se emplea para la recepción del tráfico de red a 100 Gbit/s y la otra se encarga de enviar los datos balanceados a cada uno de los equipos de análisis. Al emplear una única interfaz de la tarjeta de red para enviar todos los datos a los equipos de análisis, se hace necesario emplear un switch para gestionar la conectividad de todos los equipos. Los equipos de análisis incluyen una tarjeta con doble puerto 25 GbE Mellanox Connect-X4 NIC, dos procesadores *Intel Xeon E5-2623 v4 @2.60GHz* y cuatro discos NVMe *Intel P3600* de 800 GB. Los discos NVMe se usan

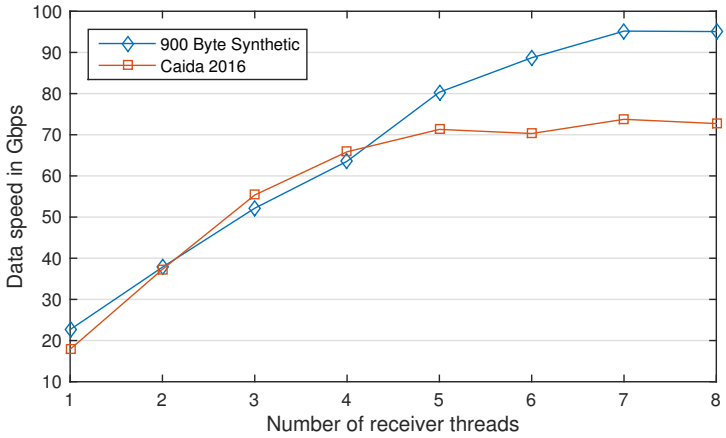


Figura 3.6. Resultados de Wormhole a 100 Gbit/s usando la traza de CAIDA y de tráfico sintético.

como almacenamiento local. Se usan ambas interfaces de la tarjeta de 25 GbE por dos razones: (i) para evitar un desbalanceo del tráfico debido a picos de tráfico en la red monitorizada, o problemas en el hash de las colas RSS; y (ii) ser capaces de enviar todos los datos procesados al equipo de visualización. Finalmente, los resultados del procesamiento de los datos se envían al equipo de visualización con el objetivo de verificar que la monitorización se realiza correctamente. Dado que las tareas de computo que debe realizar el equipo de visualización no son muy pesadas, dispone de un único procesador *Intel Xeon E5-1620 v2 @3.70GHz*, y una tarjeta 10 GbE *Intel 82599* para la recepción de los datos desde los equipos de análisis.

Una vez tenemos el banco de pruebas definido, se realizan dos experimentos con objetivo de obtener la tasa de transferencia hacia los equipos de análisis. Una de las pruebas emplea una traza de tráfico real obtenido del set de CAIDA [77]. La otra traza de tráfico se ha creado de manera sintética con paquetes de tamaño 900 bytes, que se corresponde con el tamaño medio de los paquetes de la traza de CAIDA empleada en la primera prueba. Los resultados se muestran en Figura 3.6 y puede observarse la tasa efectiva en Gbit/s. Es importante destacar que los resultados etiquetados como threads 1 a 4 se corresponden con un thread por equipo de análisis, mientras que los resultados etiquetados como threads 5 a 8 se han obtenido ejecutando dos threads por equipo, cada uno conectados a una de las interfaces de 25 GbE. En resumen, de los resultados se puede concluir que es posible analizar datos a tasa de línea en el caso de la muestra sintética, y se alcanzan los 70 Gbit/s con la traza de trafico real de CAIDA.

Una de las causas de que no sea posible alcanzar los 100 Gbit/s con la traza de tráfico real se debe al uso de micro-batches de paquetes con un mismo equipos destino. Si un equipo está saturado, esto puede tener un efecto perjudicial en la máquina de estados de TCP tanto del emisor como del receptor, produciendo unas pequeñas pausas de TCP, que finalmente pueden bloquear el sistema completo. Sin embargo, otra posible causa puede ser la naturaleza de la traza de CAIDA, o los efectos en la variación del tamaño de los paquetes cuando se procesan en memoria.

Arquitectura Big Data para la monitorización a 100+ Gbit/s

Una vez hemos definido *Wormhole* como un sistema de paso de mensajes de alta velocidad, podemos definir finalmente nuestra arquitectura de monitorización. La arquitectura completa se muestra en la Figura 3.5 y se divide en 6 fases:

Agregación de conexiones La primera fase consiste en obtener los paquetes emitidos desde los diferentes puntos de la red a monitorizar, por ejemplo, dentro de un centro de datos. Esta fase depende en gran medida de la topología de interconexión usada. Es obvio que diferentes topologías de interconexión pueden motivar diferentes técnicas de recolección. Sin embargo, para mantener una monitorización coherente, es obligatorio agregar todo el tráfico, al menos, con una granularidad de flujo, entendiendo como flujos todos los paquetes que comparten el mismo significado, como por ejemplo la misma tupla <origen, destino, protocolo>, el mismo servicio, la misma subred, etc. Este paso se presenta para completar la arquitectura, pero por simplicidad, asumiremos que disponemos de un único punto de captura 100 Gbit/s que agrega todo el tráfico de un switch o router principal de un centro de datos. Sin embargo, gracias a la arquitectura propuesta, varios puntos de captura son posibles, y el proceso de agregación puede ser considerado para futuros trabajos.

Reducción de datos sin uso Parte del proceso de monitorización pasivo consiste en identificar el tráfico y sacar información relevante del mismo. A día de hoy la mayor parte del tráfico se encuentra cifrado aportando poca información relevante más allá de sus cabeceras de red y transporte. Por tanto y con el objetivo de poder escalar sin la necesidad de incrementar el número de nodos de análisis hemos considerado añadir de forma **opcional** una fase de reducción de ancho de

banda mediante una FPGA [120]. Dado que el diseño FPGA permite reducir el ancho de banda de 100 Gbit/s a menos de 10 Gbit/s podría permitir (potencialmente) alcanzar el terabit agregando las salidas de 10 FPGAs.

Distribución La tercera fase de nuestra arquitectura se divide en dos subapartados: (I) Captura de tráfico y (II) repartición y balanceo. La forma más eficiente de realizarlos es utilizar DPDK [43] para la captura y las propias colas RSS de la tarjeta de red para realizar un reparto lo más efectivo y eficientemente posible [39]. De esta forma, necesitaremos una única aplicación de wormhole+DPDK con tantos hilos y colas como equipos de análisis tengamos en la fase de análisis, donde cada hilo leerá un paquete en DPDK y lo enviará al *Hole* de *Wormhole* correspondiente. Debido a que el tráfico capturado y balanceado tiene que ser ejecutado por un mismo proceso, esta fase puede suponer un cuello de botella si el balanceo no es óptimo.

Análisis La cuarta fase es el análisis en tiempo real paquete por paquete. Se reciben los *Worms* enviados por la fase anterior en diferentes *Holes* y la tarea consiste en: (I) reducir los paquetes a un elemento más compacto (por ejemplo, flujos) que sea almacenado para un análisis detallado posterior por una herramienta on-demand/offline y (II) sacar información útil para un análisis en tiempo real. Para validar la arquitectura propuesta se ha utilizado un analizador de flujos (Naudit DetectPro) con una capacidad de procesamiento de 15 Gbit/s por instancia. No obstante, este análisis puede ser sustituido por cualquier aplicación que un gestor de red necesite, siempre y cuando sea integrado con el sistema de paso de mensajes de *Wormhole*.

Almacenamiento e indexado La quinta fase corresponde a almacenamiento e indexado de los datos procesados. Es el punto de partida de la arquitectura Lambda, donde los datos para análisis en tiempo real, en nuestro caso las estadísticas generadas a partir de los flujos, son indexados mediante ElasticSearch[11], InfluxDB[11] o motores de indexación similares, para poder ser visualizados en tiempo real. Por el contrario, los datos "en crudo" en nuestro caso los flujos y paquetes, se almacenan en HDFS (Hadoop File System)[11] u otro sistema de almacenamiento distribuido para un posterior procesamiento más intensivo, como puede ser consultas de base de datos, algoritmos de aprendizaje automático, etc.

[11] `http://bigdata.andreamostosi.name/`

Visualización La última fase es la visualización de los datos, donde los datos indexados pueden ser mostrados mediante Grafana[11] o cualquier otra herramienta para visualización de datos. Al mismo tiempo, los datos almacenados en HDFS se encuentran disponibles para su procesamiento y posterior visualización, por ejemplo, a través de consultas usando Hive[11].

3.3. Conclusiones

Al inicio de este capítulo se introduce la problemática de la monitorización y cómo su distribución es una pieza fundamental para resolver los clásicos problemas del procesamiento mono-máquina. Una de las primeras aproximaciones para resolver esta problemática fue *Loginson*, un sistema distribuido de almacenamiento de Logs. A pesar de no haber sido el autor principal, la gran problemática presentada y toda la experiencia ganada en la colaboración de ese trabajo propició el desarrollo de Wormhole.

Gran parte de este capítulo, por tanto, se ha dedicado a la discusión y evaluación de las limitaciones de los motores de streaming actuales para su uso en la monitorización, y más concretamente a la monitorización a nivel de paquete. Para superar estas limitaciones, se ha diseñado un motor de streaming, llamado Wormhole, que incluye las capacidades de captura y comunicación necesarias para captura datos de red en enlaces de 100 Gbit/s, y repartir esos datos entre diferentes nodos de procesamiento.

Se ha mostrado que el rendimiento de Wormhole claramente supera a los motores de streaming actuales, siendo capaz de repartir hasta 24 Gbit/s de datos, con una latencia muy similar al mejor motor de streaming analizado, únicamente con un único proceso. Además, la herramienta desarrollada permite escalar linealmente con el número de equipos de procesamiento, hasta casi alcanzar los 100 Gbit/s, existiendo una limitación debida al balanceo de la carga, así como sobrecargas asociadas a TCP. Es posible que con un poco más de trabajo se puedan identificar más claramente, en particular en el caso de usar tráfico real, pero los resultados actuales nos permiten cumplir con los objetivos propuestos, e incluso despejar el camino para permitir el análisis de datos a Terabit/s.

Finalmente, se presenta una arquitectura Big Data de monitorización a 100 Gbit/s, donde el motor desarrollado, Wormhole, es una parte importante. Esta arquitectura propuesta y probada en esta tesis es posiblemente la primera aproxima-

ción que usando soluciones Big Data permite abordar la monitorización de redes de 100 Gbit/s e incluso plantear la posibilidad de funcionar en redes de mayor velocidad, cubriendo todos los aspectos de la monitorización, desde la agregación de los puntos de red hasta la visualización por parte de un gestor de red.

4

Análisis y visualización

El capítulo anterior termina con la descripción de la arquitectura Big Data propuesta como objetivo de esta tesis. Tras poner en marcha la arquitectura propuesta, disponemos de la información extraída de los datos de red, en este caso, los flujos de red y los paquetes, almacenados en un sistema de ficheros distribuido. En este capítulo se proponen una serie de procesamientos sobre los datos con el objetivo de poder manejarlos, enriquecerlos y darles una utilidad para el análisis y extracción de información que puede ser útil para la gestión de la red, la seguridad, etc.

4.1. Análisis offline: Hive y Hadoop

La primera opción a la hora de elegir el tipo de herramientas que se pueden utilizar para procesar los datos almacenados depende directamente del sistema de ficheros seleccionado para su almacenamiento. Si escogemos el más famoso, HDFS, es obviamente, el conjunto de herramientas creadas alrededor del proyecto Apache Hadoop [121] las que debemos utilizar.

El proyecto Apache Hadoop [121] nació como alternativa libre a las versiones MapReduce [122] y GFS [123] de Google. El núcleo consta de un sistema de fiche-

ros distribuído, *Hadoop Distributed File System* (HDFS) [100] y del paradigma de programación *MapReduce*, conociéndose en su última versión estable como *YARN* o *MapReduce 2.0* [124]. Dichas herramientas permiten paralelizar el análisis de los datos de una forma transparente y eficiente, llevando la computación a los datos. El proceso es escalable y robusto, ya que si falla cualquier nodo, un disco o la red, el sistema continúa funcionando.

Sobre el núcleo de Hadoop surgen multitud de herramientas, comúnmente llamadas su ecosistema. Entre las herramientas más conocidas se encuentra Apache HBase para el almacenamiento estructurado de los datos y su posterior consulta, Apache Pig para el análisis a alto nivel, o Mahout para realizar aprendizaje automático. Este ecosistema se está extendiendo día a día, y es usado por grandes empresas como *eBay*, *Facebook*, *Google* o *LinkedIn* [125]. Hadoop ya forma parte de algunas plataformas de computación, como *Elastic Compute Cloud* (EC2) de *Amazon*, *CDH* (*Cloudera's software distribution containing Apache Hadoop*) o el *IBM InfoSphere BigInsights* [126].

Hadoop fue diseñado para el procesamiento por lotes, con aplicaciones en diversos ámbitos como la minería de datos, la indexación de páginas web o el análisis de *logs*. Por ello, la lectura de ficheros de texto plano está muy simplificada en Hadoop. Incorpora también una clase que permite leer cómodamente ficheros binarios siempre que estén en un formato determinado. Sin embargo, no está preparado para manejar el formato estándar de almacenamiento de paquetes de red, PCAP, directamente, pues en principio no sabe cómo extraer, de forma individual, los paquetes que contienen. Una forma de resolver esto es realizar la conversión de formato para cada PCAP. No obstante, esto requiere demasiado tiempo y espacio, por lo que en trabajos como los de RIPE [13] o Y. Lee [108] han creado sus propias clases Java que lidian con este problema y permiten abrir ficheros PCAP. El trabajo de RIPE ha sido uno de los primeros en mezclar el mundo de la computación distribuida y Hadoop con el mundo de las redes y su monitorización. Su implementación permite extraer la información de los paquetes, proporcionando una API con la información de los diversos niveles OSI. No obstante, esta implementación solo permite diseccionar los protocolos a nivel de aplicación DNS y HTTP, para lo cual utiliza bibliotecas de terceros. Por otro lado, en [108] Y. Lee presenta un trabajo un sistema de monitorización capaz de procesar, mediante una API diferente, tráfico IP, TCP, HTTP y registros NetFlow. Por último, en [127], Y.Lee presenta una heurística por la cual es posible fragmentar ficheros PCAP en diver-

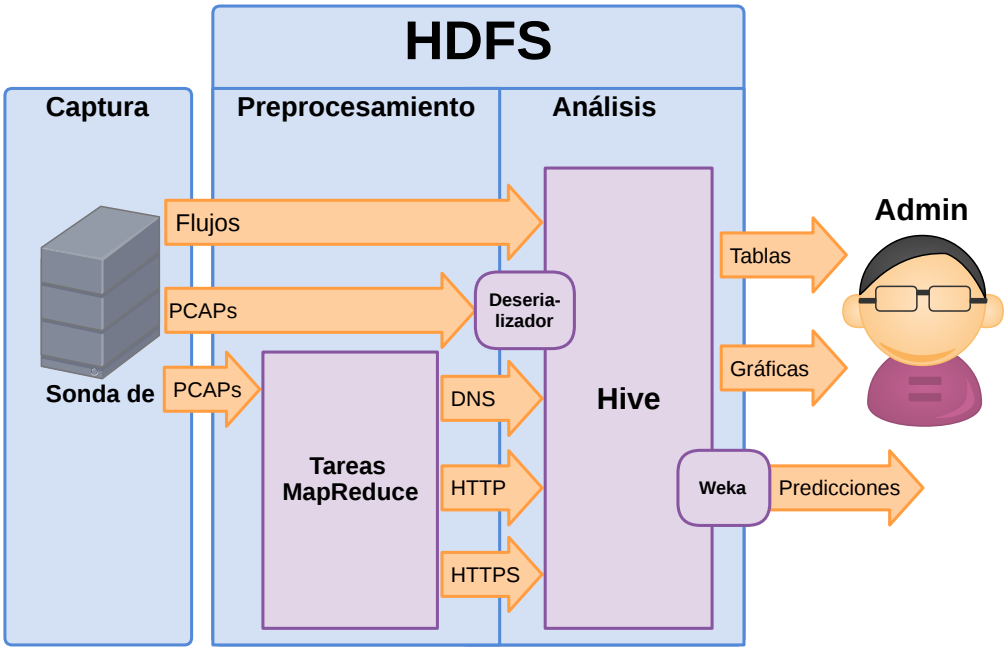


Figura 4.1. Arquitectura del sistema propuesto

sos bloques, de modo que Hadoop pueda procesar un único archivo de varios Gigas o Teras entre diferentes nodos.

4.1.1. Procesamiento de ficheros PCAP en Hadoop

El objetivo es conseguir que las herramientas de Hadoop puedan manipular los ficheros de flujos, pues contienen información resumida del tráfico de red, y los ficheros PCAP, por ejemplo, para realizar un análisis más específico sobre protocolos como DNS, HTTP y HTTPS, para lo que se será necesario crear tareas *MapReduce*. Si aun así se necesita información más detallada, se debe desarrollar un deserializador que transforma los paquetes en datos estructurados. De este modo, los archivos PCAP resultan transparentes para cualquier herramienta del ecosistema Hadoop.

Para el procesamiento de los ficheros PCAP se parte de la implementación de RIPE [13], sobre la cual se han realizado varios cambios, fundamentalmente en un intento de mejorar el rendimiento a costa de una pérdida de generalidad y flexibilidad. En primer lugar, la implementación de RIPE utiliza bibliotecas de terceros para procesar los protocolos DNS y HTTP que descartan todos los paquetes truncados. En nuestro caso, la segunda fase de la arquitectura propuesta en el capítulo anterior, proponía la posibilidad de truncar los paquetes, por lo que no es

aceptable que se descarten los paquetes truncados. Por ello, se han desarrollado nuevos disectores desde cero de una forma más eficiente, guardando únicamente la información que se considera imprescindible.

Una ventaja de la implementación de RIPE es su flexibilidad y generalidad, la cual se obtiene gracias a la forma de guardar la información: en lugar de almacenar los campos en una estructura estática con registros predeterminados, la implementación de RIPE utiliza un array asociativo, cuyos índices vienen determinados por los nombres de los respectivos campos. Esto permite adaptar dinámicamente los campos de las estructuras a la información específica que contiene un determinado paquete o flujo. Un ejemplo práctico es la disección de HTTP: la cabecera tiene una gran variedad de campos posibles que además evolucionan con el tiempo [128]. En nuestro caso de estudio conocemos a priori todos los campos que nos interesan, por lo que una implementación como esta supone una pérdida de eficiencia tanto de memoria como de tiempo. Por este motivo nuestra implementación almacena la información en clases solo con los campos apropiados.

Otra ventaja de su implementación es la posibilidad de añadir cómodamente nuevas clases que amplíen el repertorio de protocolos que pueden ser diseccionados sin modificar su código. Esta flexibilidad se consigue indicando al principio, mediante un parámetro de Hadoop, el nombre de la clase que se va a utilizar para procesar la capa de aplicación. El principal inconveniente de esta aproximación es que solo se puede especificar un protocolo, por lo que si se quieren analizar varios se tendrá que codificar un programa *MapReduce* por cada protocolo. En nuestro caso se ha mantenido dicho diseño y, aprovechando la flexibilidad que proporciona se ha añadido una clase que disecciona HTTPS, dado que una gran parte del tráfico web de hoy en día se encuentra protegido por este protocolo. Sin embargo, con vistas a mejorar la eficiencia, habría que crear una única clase capaz de analizar todos los protocolos a nivel de aplicación simultáneamente.

Otro cambio importante sobre el diseño de RIPE se ha hecho en el retorno de los paquetes: mientras que su iterador devuelve todos los paquetes que lee, el nuestro no devuelve aquellos que no contienen información sobre el protocolo indicado al inicio. Como esto se va a utilizar para el análisis de los protocolos de nivel de aplicación, es innecesaria la obtención de paquetes que no contienen esta información.

Por último, en la implementación original de RIPE, se aprecia una gran pérdida de rendimiento ocasionada por el reensamblado de los paquetes al lidiar con la fragmentación IP y la reconstrucción de flujos TCP unidireccionales. Debido a

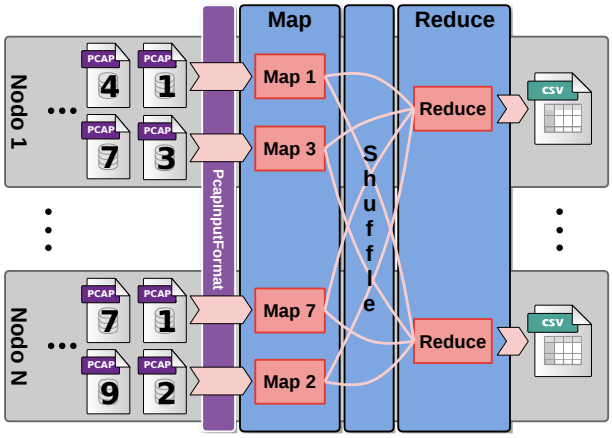


Figura 4.2. Ejemplo de ejecución de un programa *MapReduce* sobre archivos PCAP.

que nuestro tráfico se encuentra truncado y solo nos interesan los primeros bytes de cada flujo, en la mayoría de los casos, podemos prescindir del reensamblado.

Disectores Una vez se han enumerado las diferencias con la aproximación inicial de RIPE, es posible comenzar a explicar los detalles de la implementación de los diferentes disectores de tráfico. En primer lugar, se dará una visión general de su funcionamiento, que en gran medida es la forma de actuar del paradigma *MapReduce*. En la Fig. 4.2 se muestra un diagrama simplificado de las fases de la ejecución de los disectores realizados. El planificador de Hadoop asigna qué archivo debe procesar cada nodo, de forma que se lance una tarea *Map* por archivo. Gracias a la clase `PcapInputFormat`, las tareas *Map* reciben solo los paquetes del protocolo especificado, que procesan por separado y emiten uno o varios pares (clave, valor) por cada paquete. En la fase de *Shuffle*, interna de Hadoop, los pares con la misma clave son agrupados y enviados al nodo adecuado para poder iniciar la última etapa. En esta última fase de *Reduce*, se procesan por separado cada uno de los grupos de valores que comparten la clave, generándose una o varias líneas de texto de la salida.

En las siguientes subsecciones, se comentan en detalle cada uno de los disectores realizados, así como la salida que estos generan.

Disector HTTP El disector HTTP empareja las peticiones HTTP con sus respectivas respuestas. La salida de este programa es un fichero CSV con la informa-

ción de cada flujo HTTP. Los campos contenidos en este fichero son: las direcciones IP, los puertos, los tiempos de inicio de la petición y de llegada de la respuesta, el método HTTP usado, el host de destino, el recurso de la URL pedida, y por último el código y la descripción de la respuesta. Retransmisiones o peticiones sin respuesta son descartadas.

De la obtención de los campos HTTP se encarga la clase `HttpPcapReader`, implementada sobre la base de RIPE. Esta recibe la carga útil del protocolo de nivel de transporte, e identifica el protocolo de aplicación utilizando DPI y simples expresiones regulares. Al encontrarse los paquetes truncados, es posible que ciertas partes de la cabecera no estén presentes y otras se encuentren incompletas. Gracias a esta clase, las tareas *Map* procesan cada paquete identificado como HTTP de uno en uno. Para emparejar cada petición con su respuesta, estas tareas generan claves a partir de las direcciones IP y los puertos. Es importante recalcar que, aunque en número de *ACK* de la respuesta se puede calcular a partir de la petición, si la petición está fragmentada a nivel TCP no se podrá realizar este cálculo, ya que no se realiza el reensamblado. En los valores se guardará el resto de campos necesarios para generar la salida anterior, incluyendo el *ACK* en caso de las respuestas y el siguiente número de secuencia en el de las peticiones. Finalmente, las tareas *Reduce* del disector ordenan los valores antes mencionados, generando una línea del fichero de salida por cada conexión HTTP, incluyendo, por tanto, la información relevante de su petición y su respuesta.

Disector DNS Muchas de las peticiones HTTP capturadas no contienen el campo *Host* debido al truncado de los paquetes en la fase de captura. Consultar el nombre de todos los servidores mediante un script que acceda al DNS de la UAM es poco eficiente, y además dado el número de consultas por segundo a resolver, podría ser considerado un DDoS, causando un posible bloqueo temporal por parte del servidor. La solución propuesta es una búsqueda en el tráfico DNS disponible para identificar el nombre de los servidores a partir de su dirección IP. El objetivo de este disector es crear una tabla con todos los nombres consultados hasta el momento, junto con su resolución IP.

Sobre la base de RIPE se ha implementado la clase `DnsPcapReader`, que obtiene todas las respuestas a las consultas de tipo *A* (dirección de un host) y de clase *IN* (Internet). Aunque el registro *CNAME* es también importante para la resolución final, actualmente se está trabajando para soportar este tipo de respuestas y así ampliar el número de direcciones IP almacenadas. Todos los demás tipos de res-

Sistema		RAM (GB)	Disco (TB)	CPU	Cores
Hadoop	1	32	15,2	1x Intel Xeon L5408 @ 2,13GHz	4
	2	32	15,2	1x Intel Xeon L5408 @ 2,13GHz	4
	3	256	12,1	4x Intel Xeon E7-4830 @ 2,13GHz	32
	4	64	20,6	2x Intel Xeon E5-2620 v3 @ 2,40GHz	12
	5	64	20,6	2x Intel Xeon E5-2620 v3 @ 2,40GHz	12
Serie		32	RAID0:9x3	2x Intel Xeon E5-2630 @ 2,6GHz	12

Tabla 4.1: Características de los sistemas de pruebas

puestas son ignoradas, así como todas las consultas, puesto que lo único que nos interesa es la resolución de las direcciones IP. Usando la clase `DnsPcapReader`, las tareas *Map* reciben las consultas DNS mencionadas y emiten, para cada una de sus respuestas, una cadena de texto que consta únicamente del nombre consultado y la dirección IP resuelta. Las tareas *Reduce* se encargan de eliminar las cadenas duplicadas.

Disector HTTPS Cómo resultado del aumento de la seguridad en los sistemas actuales, la mayor parte del tráfico se encuentra cifrado. El protocolo por excelencia en este campo es HTTPS, por lo que se ha considerado procesar también este protocolo. Aunque los datos se encuentran cifrados, el nombre del servidor se puede obtener fácilmente examinando la negociación del protocolo TLS. Con este fin se ha implementado la clase `HttpsPcapReader` sobre la base de RIPE, que considera únicamente los mensajes de tipo *Hello*, y obtiene de ellos el nombre del servidor. Tras esto, la tarea *MapReduce* genera una línea por paquete procesado con las direcciones IP de las máquinas que intervienen, y el nombre del servidor obtenido.

4.1.2. Pruebas de Rendimiento

Antes de comparar los resultados del rendimiento de la herramienta presentada, en la Tabla 4.1 se detallan las características de los sistemas que se han utilizado para evaluarlo. Como se puede observar, se dispone de dos sistemas: el clúster Hadoop para desarrollo que consta de cinco nodos, y un ordenador de altas prestaciones donde se han realizado pruebas en serie (a partir de ahora, nodo serie).

El clúster de Hadoop de desarrollo consta de 8 nodos conectados mediante una red de 1 Gpbs, de los cuales solo los 5 nodos más potentes realizan la computación.

Por otro lado, para evaluar nuestros resultados se ha tenido en cuenta el rendimiento obtenido por Y. Lee [108], que dispone de un clúster de 30 nodos conectados mediante una red de 1 Gbps, cada uno con 8 cores a 2,93 GHz.

Para evaluar el rendimiento de los tres disectores explicados, se ha comparado su velocidad de procesamiento con 1) un test de lectura incluido en la distribución de Hadoop, 2) una versión en serie escrita en C del disector HTTP [129] y ejecutada en el nodo serie, y 3) los resultados presentados por Y. Lee. La entrada de todas las pruebas realizadas ha sido la misma, 1 TB de tráfico real obtenido de la red de los laboratorios de la Escuela Politécnica Superior de la UAM. Nótese que la comparación con las pruebas de Y. Lee no es totalmente justa, pues ni el tráfico ni el procesamiento realizado son los mismos. No obstante, sus diferentes análisis alcanzan rendimientos parecidos, y para la comparativa se han usado sus mejores tiempos.

En la Tabla 4.2 se muestra el rendimiento de todas estas pruebas, medido en Gbps y en Mpps (Millones de paquetes por segundo), así como su eficiencia medida en Gbps por core utilizado. Se puede observar que el rendimiento de cada core del sistema implementado dobla el de los cores de Y. Lee, quienes a su vez superan el rendimiento de RIPE [108].

Por otro lado, el test de lectura de Hadoop, que consiste en ejecutar el benchmark TestDFSIO disponible con Hadoop, tiene un rendimiento mucho menor que el de los disectores, que deben procesar los datos además de leerlos. Esto se debe a que el test de lectura permite la división de los archivos en 8 bloques de 128 MB, por lo que el planificador de tareas necesita mucho más tiempo para completar su trabajo, convirtiéndose en el cuello de botella. Reduciendo el tamaño de la entrada se ha comprobado que los tiempos de lectura y de los disectores se igualan, por lo que la mayor parte del tiempo se invierte en la lectura de los archivos. Finalmente, el principal hándicap del sistema propuesto es su heterogeneidad, puesto que el número de cores por máquina varía desde 4 hasta 32. Debido a esto, algunos nodos terminan sus tareas antes que otros, viéndose forzados a pedir a otros nodos nuevos archivos PCAP que procesar, lo que acaba por provocar una penalización por el aumento en las comunicaciones.

4.1.3. Análisis

Mediante HiveQL, el lenguaje de Hive basado en SQL, un administrador de red puede crear sus propias consultas que, de forma automática y transparen-

Tabla 4.2: Rendimiento de cada sistema y de cada core

Programa		Gbps	Mpps	Gbps/core
Hadoop	Disector HTTP	7,13	3,96	0,11
	Disector HTTPS	7,41	4,12	0,12
	Disector DNS	7,72	4,29	0,12
	Test de lectura	4,98	-	0,08
Disector HTTP en serie		4,21	2,34	4,21
Versión de Y. Lee [108]	5 nodos	1,9	-	0,05
	30 nodos	14,0	-	0,06

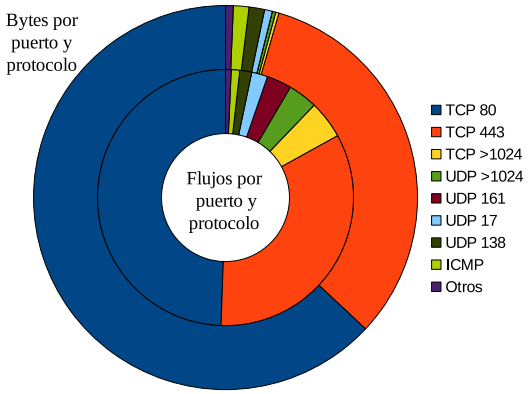


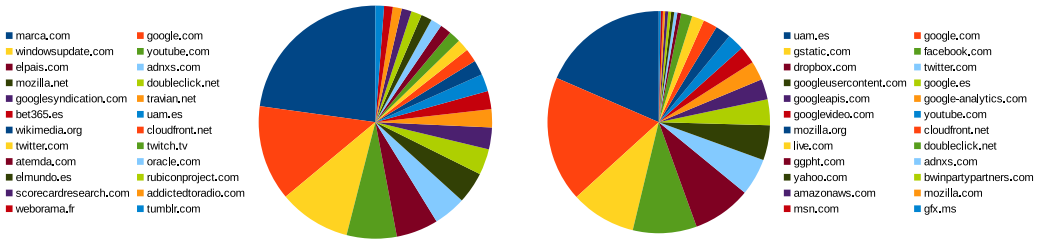
Figura 4.3. Porcentaje de Bytes y de flujos de los puertos y protocolos más comunes

te, son traducidas a tareas *MapReduce*. Se dispone así de una forma cómoda de realizar un análisis de grandes cantidades de datos.

Sobre los archivos de flujos pueden realizarse las consultas más generales, es decir, las que requieran únicamente campos como direcciones IP, puertos, número de paquetes o duración del flujo. Por ejemplo, se han creado series temporales que muestran en cada momento el grado de utilización de la red, de una subred determinada, o incluso de cada uno de los ordenadores. Gracias a este tipo de consultas se han podido detectar ordenadores averiados, que no generan ningún tipo de tráfico a lo largo de un periodo largo de tiempo, así como los ordenadores que se han dejado encendidos alguna noche. Otros ejemplos relevantes son el estudio de los puertos y protocolos usados, ya que con esto se puede comprobar si el cortafuegos está funcionando correctamente, así como analizar las conexiones iniciadas desde el exterior con el fin de detectar posibles ataques. En la Fig. 4.3 se muestra el uso de cada puerto y protocolo. El anillo exterior representa la cantidad de bytes, mientras que el anillo interno, muestra el número de flujos.

En el caso particular presentado, al tratarse de una red universitaria, se pueden tratar también de optimizar los recursos de red. Para ello se ha estudiado el consumo de ancho de banda de cada laboratorio en función de las horas del día, si hay o no clase y la asignatura que se imparte en cada momento. Es posible también estimar el número de ordenadores activos simultáneamente en días especiales como huelgas, periodos de exámenes o vacaciones con el fin de minimizar el personal o el número de laboratorios abiertos necesarios sin perjudicar a los usuarios.

Utilizando los archivos resultantes del preprocesamiento se puede realizar un análisis más específico, que tenga en cuenta campos de los protocolos DNS, HTTP o HTTPS. Usando los datos DNS se puede rellenar el campo *Host* de los registros HTTP truncados en los que se perdía esta información. Con los datos HTTP y HTTPS se han realizado diferentes tipos de consultas y análisis. Gracias a las consultas más simples, es posible obtener las URLs más populares, así como las páginas más visitadas durante cada asignatura. El top de URLs suele ser una métrica común a la hora de realizar diversos análisis red, ya que aporta interesante información acerca de utilización de la misma por parte de los usuarios. En nuestras pruebas, es observable que los usuarios dedican gran parte del tiempo en los laboratorios al ocio (ver Fig. 4.4). No obstante, a pesar de las ventajas que ofrece este tipo de análisis, realizarlo supone un proceso computacionalmente costoso, por lo que suele ser uno de los elementos más comparados.



(a) Por protocolo HTTP **(b)** Por protocolo HTTPS

Figura 4.4. Top de peticiones a host por protocolo

Por último, si durante el análisis fuesen necesarios campos no disponibles en alguno de los archivos preprocesados, sigue siendo posible realizar las consultas directamente sobre los archivos PCAP mediante un deserializador.

4.2. Análisis profundo de paquetes (DPI)

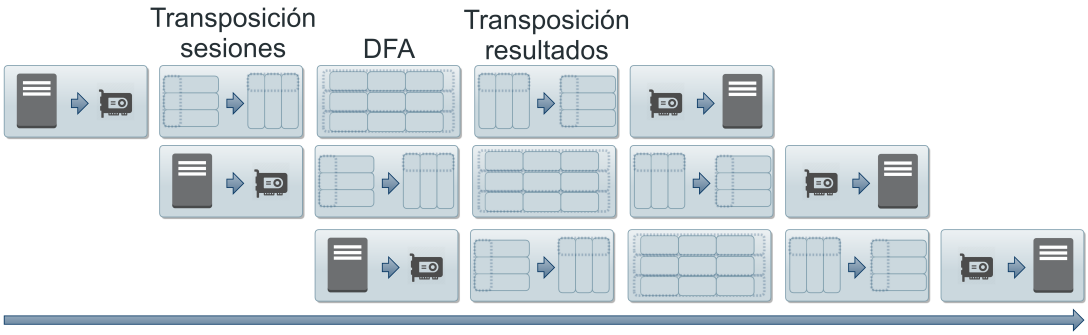


Figura 4.5. Arquitectura del sistema DPI en la plataforma GPU

En la sección anterior se han visto las posibilidades que ofrecen las herramientas Big Data para el análisis de los datos obtenidos de la red, una vez se encuentran almacenados en HDFS. Se ha mostrado que es posible realizar disectores en aplicaciones MapReduce para obtener información de protocolos bien conocidos como DNS o HTTP. Sin embargo, por las redes de comunicaciones también se envía información de aplicaciones que no quieren ser descubiertas o analizadas. Por ejemplo, en determinados casos los protocolos son ejecutados sobre el puerto 80 con el fin de atravesar posibles restricciones impuestas por un cortafuegos.

La necesidad de identificar este tráfico, incluso en tiempo real, es una obligación para un determinado grupo de aplicaciones. Un ejemplo son las herramientas de *detección de intrusiones* (IDS) tales como *Snort* [130] o antivirus, *ClamAV*. Por tanto, se hace necesario aplicar técnicas de clasificación con objetivo de identificar este tipo de tráfico y poder analizarlo.

Los métodos de clasificación de tráfico son variados, entre los más utilizados destaca la *clasificación por puerto* a nivel de transporte y DPI (*Deep Packet Inspection*). Este último consiste en inspeccionar la carga útil de los paquetes transferidos en lugar de buscar un campo fijo. Ambos casos pueden ser combinados con la selección arbitraria de los paquetes a analizar, aplicando métodos tan populares como Naïve Bayes o los estimadores Kernels que, con un 10 % del total del tráfico contemplado pueden llegar a obtener una precisión del 96 % [131]. Una limitación importante que no debe obviarse es la imposibilidad de aplicar tales técnicas sobre protocolos cifrados, en los cuales la información no podrá ser reconstruida.

Existen multitud de trabajos que comparan las ventajas y contrapartidas entre los diferentes métodos. La desventaja más criticada al uso de DPI radica en

su alto coste computacional [132], el cual puede ser paliado si se utilizan las arquitecturas adecuadas, generalmente aceleradores hardware. Sin embargo, DPI ofrece una mayor precisión en la clasificación que el resto de métodos. Por ello, se sigue motivando como método de clasificación de tráfico en sistemas críticos donde no existe tolerancia a fallos. El primer paso para clasificar a altas velocidades reside en reensamblar el tráfico subyacente; el segundo paso, buscar coincidencias basándose en reglas sobre los datos. La tarea de recomposición ha sido explicada detalladamente en trabajos como [6]. La forma más común de expresar una regla de DPI consiste en su formulación como expresiones regulares PCRE (*Perl Compatible Regular Expression*), como las que proporciona *l7filter*. Estas expresiones son evaluadas en entornos de alto rendimiento haciendo uso de *Autómatas Finitos Deterministas* (DFAs) [133] que simplifican la evaluación tras su previa construcción.

En el mismo documento [6] se contempla el problema de capturar la información a nivel de sesión a 10 Gbps con tarjetas de red multigigabit de Intel, la transferencia de los datos con un driver optimizado para el experimento y una unidad de procesamiento gráfico que clasifica la carga útil. En un entorno CPU se hacen patentes las limitaciones cuando se pretende analizar todo el tráfico y realizar su clasificación. [134] expone cuál es la penalización de emplear DFA y por ello justifica las alternativas basadas en aceleradores hardware.

Partiendo de estos precedentes, se propone implementar una herramienta de clasificación de paquetes que haga uso de las 145 firmas proporcionadas por la utilidad *l7filter*. En una primera aproximación se quiere partir de los resultados de la sección anterior y hacer uso Hadoop, Hive y MapReduce. Sin embargo, en un entorno CPU se hacen patentes las limitaciones cuando se pretende analizar todo el tráfico y realizar su clasificación [134], y por ello se justifica el uso de aceleradores hardware. Por tanto, se considera realizar el desarrollo para GPU, FPGA y Xeon Phi, de modo que sea posible acelerar las tareas MapReduce en Hadoop, ya que se ha demostrado que es posible la integración de estos aceleradores HW en el ecosistema Hadoop [135–137].

4.2.1. Plataformas y adaptación

Cada una de las firmas de la herramienta *l7filter* se preprocesan generando diferentes DFA, que posteriormente son reutilizados en cada una de las diferentes plataformas. Estos autómatas pueden contener a su vez un número variable de

firmas, minimizando el número de máquinas de estados. Hay que tener en cuenta que el número de estados crece proporcionalmente con el número de firmas, así como los recursos y memoria que utilizan las FSMs ("Finite-State Machines").

4.2.2. Intel *Xeon Phi*

Los productos bajo la etiqueta *Xeon Phi* conforman una familia de aceleradores caracterizada por la inclusión de un alto número de procesadores y una elevada efectividad en la realización de operaciones en coma flotante. La inclusión de una versión adaptada del sistema operativo Linux frente a una alternativa aplicada directamente sobre los recursos hardware (bare metal) facilita que diseños existentes sean portados a la arquitectura con relativa sencillez.

Para el experimento se somete a evaluación el *modelo 31S1P* de la serie 3100, con un total de 57 procesadores de propósito general, más un procesador adicional destinado a tareas de gestión y sincronización, cada uno de los cuales puede ejecutar hasta 4 hilos de manera simultánea. La frecuencia del reloj se eleva a los 1.1 GHz mientras que la memoria principal es de 8 GB. La memoria caché de nivel 2 es de 28,5 MB virtuales y está formada por 512 KB de cada procesador conectados en una topología de anillo doblemente enlazado con las unidades de procesamiento (PU) contiguas. Adicionalmente, disponen de una unidad procesamiento vectorial (VPU) encargada de aplicar operaciones SIMD (Single Instruction, Multiple Data) sobre vectores de 512 bits.

Partiendo de una versión serie del algoritmo DPI, se aplican una serie de estrategias para adaptar el algoritmo y obtener un mejor rendimiento: paralelizar el número de firmas que se aplican simultáneamente sobre una sesión de red, y paralelizar en el número de sesiones sobre las que se pretende aplicar DPI. La complejidad reside en el patrón de acceso a memoria en detrimento de la capacidad de computación. Para la comprobación simultánea de firmas, no existen dependencias, por esta razón se aplicará paralelización de grano más grueso. Para el aprovechamiento de la unidad de procesamiento vectorial (VPU), se forzará el uso de operaciones vectoriales. No debe olvidarse que no es el principal objetivo contabilizar el tiempo de creación de las matrices de transición puesto que dicha acción únicamente se aplica en primera instancia y nunca para la clasificación de los flujos.

Como detalle adicional, para una lectura óptima de los datos de entrada, es de obligado cumplimiento la transposición de las sesiones tras su lectura de memoria.

Se requiere que los datos de una sesión particular no estén consecutivos, sino que, por el contrario, estén agrupados en el mismo número que cantidad de *sesiones paralelas* se estén contemplando.

4.2.3. GPU

Dentro de la computación sobre tarjeta gráfica o GPU, Nvidia es uno de los fabricantes de referencia y junto con la API de programación *CUDA* ha sido la opción elegida para implementar el coprocesador utilizando *GPGPU* (Programación de propósito general utilizando GPU). La tarjeta gráfica seleccionada para las pruebas es la *Tesla M2090*, la cual dispone de 512 cuda cores a 1.30 GHz y 6 GB de memoria GDDR5. Los cuda cores de esta gráfica se encuentran agrupados en 16 MPs (*multiprocessors*). Cada MP, comparte la lógica de control entre todos los cores que pertenecen a este MP. Esto limita fuertemente la capacidad de procesamiento de una tarjeta gráfica, pues cada grupo de 32 hilos debe ejecutar la misma instrucción en cada ciclo, ya que, en caso contrario, cada hilo que se ejecute sobre estos MP pasa a ser ejecutado de forma serie. Para la evaluación de DFAs, al no existir bifurcaciones en el código, una GPU puede paralelizarlo de forma adecuada sin entrar en problemas de serialización.

Para poder hacer una comparativa del rendimiento de clasificación en GPU, se ha decidido partir del trabajo realizado en [138], al cual se le han aplicado una serie de optimizaciones. Se ha sacrificado latencia construyendo un pipeline (ver Figura 4.5), que consta de las siguientes etapas: transferencia de datos, transposición de la carga útil de las sesiones, aplicación del DFA, transposición de los resultados y devolución de la información.

4.2.4. Virtex 7 VC709 de Xilinx

La plataforma hardware reconfigurable seleccionada para la aplicación de DPI es la placa VC709, capaz de lograr un alto rendimiento gracias a las FPGAs Virtex 7 VX690T de Xilinx. Las principales motivaciones para su elección son la inclusión de 8 líneas PCIe de generación 3.0 en conjunción con el soporte para el core DMA desarrollado por *Northwest Logic*. Un total de 4 módulos SFP+ (4 transceptores 10 GbE) y 8 GB DDR3 SODIMM aseguran los mecanismos necesarios para poder trabajar a tasa de línea en enlaces multigigabit. Es decir, una arquitectura híbrida (CPU y FPGA) donde los datos capturados por la interfaz de red sean procesados y transmitidos al anfitrión es viable sobre esta plataforma [139].

El algoritmo DPI debe adaptarse para el dispositivo. El alojamiento de las tablas de transiciones en memoria BRAM es inviable sin dividir el proceso en distintas etapas (pueden llegar a contener cientos de estados distintos dependiendo de la complejidad de la tabla). Supóngase el caso donde existe una tabla con 1000 elementos, el tamaño del alfabeto es 256 (distintas posibilidades para 8 bits) y que cada entrada tiene 16 bits destinados a redireccionar el siguiente estado y otros tantos para indicar cuál de todas las coincidencias ha ocurrido (tabla de transiciones donde se evalúan múltiples firmas). En tal caso, el tamaño total es de 1000 KB. La capacidad típica de una memoria BRAM es de 4.5 KB (36 Kbits) por lo que para el caso particular expuesto anteriormente se necesitarían 223 memorias de este tipo. Asumiendo que hay un total de 1470, la aplicación no ofrecerá una alta escalabilidad.

Dado que en todo el proceso de identificación se está construyendo una máquina de estados determinista a partir de la PCRE, la aproximación discurre por generar código HDL a partir de este objeto. Se generan tantos bloques lógicos como firmas se pretendan evaluar. Cada bloque incluye una FSM capaz de consumir un byte de la cadena de entrada en cada ciclo de reloj. Tras consumir una sesión se tiene el resultado indicando la posible coincidencia. Siempre que el área y las restricciones de tiempo debidas al rutado lo permitan se podrán agregar nuevos componentes comparadores sin aumentar la penalización temporal. La arquitectura empleada para la implementación FPGA se muestra en la Figura 4.6. Este módulo principal puede ser instanciado tantas veces como sesiones se deseen evaluar paralelamente.

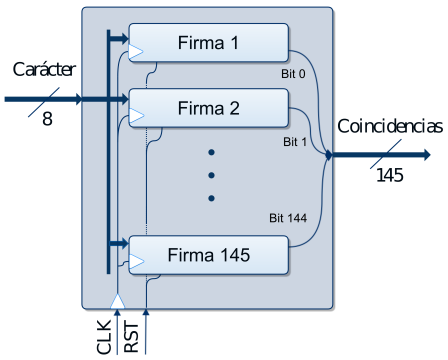


Figura 4.6. Arquitectura del sistema DPI en la plataforma FPGA

La gran ventaja de la FPGA es la capacidad de comprobar múltiples firmas DPI simultáneamente, y la facilidad de implementación de éstas, al margen de su complejidad.

4.2.5. Presentación de resultados

En este apartado se exponen los tiempos recogidos tras la evaluación empírica de la solución en las distintas plataformas. En todos los casos se utilizan los mismos datos de entrada, obtenidos a partir de una traza en formato PCAP de elaboración propia. El tamaño original del fichero se sitúa en los 50 GB de datos, a partir del cual se han reconstruido las sesiones a nivel de la capa de transporte y con un límite en el tamaño de la carga útil de 256 bytes. Tamaño que en [140] se justifica como suficiente.

4.2.6. Resultados *Xeon Phi*

No se pueden considerar los resultados de aceleración en la plataforma hardware sin evaluar a su vez la mejora ofrecida al rediseñar el código de manera más escalable. Es por ello que en este apartado se evalúan simultáneamente las mejoras tanto en CPU como en la unidad de coprocesamiento. El equipo considerado para realizar las pruebas cuenta con un procesador Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz, 8GB de memoria DDR3 a una frecuencia de 1600 MHz divididos en dos bancos y un disco duro mecánico de 512GB a una velocidad de rotación de 7200RPM. Sobre este mismo equipo se conecta la unidad *Xeon Phi*, sobre el puerto PCIe de tercera generación.

El tiempo de ejecución en un solo core de la CPU, que denominaremos tiempo serie, se sitúa en los 421 segundos frente a los 3792 segundos de *Xeon Phi*. La utilización de procesadores más sencillos arquitectónicamente junto con una frecuencia más comedida explica el resultado tan desfavorable para *Xeon Phi*.

En la Figura 4.7 se compara el efecto de añadir hilos al procesamiento. La versión de referencia para calcular la aceleración es el tiempo serie de la versión CPU. Debido a que el problema no es completamente paralelizable (entrada/salida de datos), la Ley de Amdhal define una cota superior para que la eficiencia no sea idónea. Mientras que la versión CPU consigue aportar una aceleración del 6,1, *Xeon Phi* únicamente del 1,9. Si se compara la aceleración obtenida por este último respecto a la versión inicial en la misma arquitectura el speedup es de

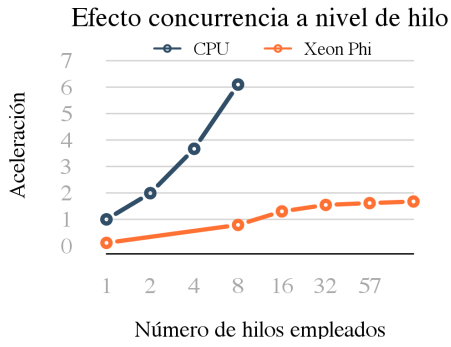


Figura 4.7. Aceleración en función del número de hilos (sin uso de la VPU) para el total de firmas.

15,1. Como conclusión, incrementar el número de hilos no va a provocar una mejora destacable en el rendimiento de la aplicación. En la Figura 4.8 se tienen las mejoras sobre la versión serie inicial al aplicar la configuración óptima de hilos y variar el número de sesiones (explotación de la VPU).

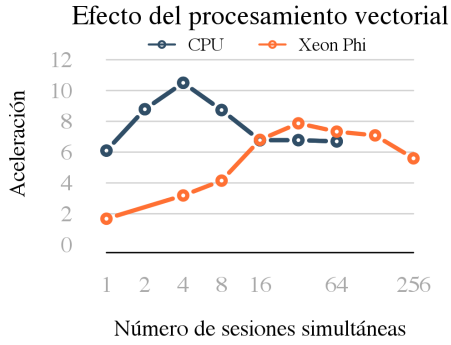


Figura 4.8. Aceleración de la VPU para el total de firmas y mejor configuración de la Figura 4.7.

Al comenzar a usar operaciones vectoriales se produce un aumento del rendimiento. En el momento en el que el número de sesiones paralelas no puede ser albergado en caché, el rendimiento decrece nuevamente. La aceleración respecto a la versión inicial es del 10,50 en CPU y del 7,87 en *Xeon Phi* para 4 y 32 sesiones respectivamente.

El problema observado es el mal rendimiento ofrecido por la unidad hardware en la ejecución serie. En total se ha conseguido una reducción del tiempo de ejecución en un factor de 70,89, una aceleración muy loable pero que, debido a la gran diferencia con la CPU, el aplicar técnicas de paralelismo en ambas arquitecturas es muy difícil que *Xeon Phi* se termine imponiendo al no utilizar operaciones en coma flotante.

En cualquier caso, se pone de manifiesto la necesidad de aplicar las mejoras introducidas en el código a la hora de convertirlo más paralelizable en la versión original ya que muchas veces las optimizaciones que se aplican para portarlo a una tecnología particular no son posteriormente anexadas.

4.2.7. Resultados GPU

La eliminación de la fase del pipeline de *reducción* en [138] facilita la obtención de un mejor rendimiento. El propio Kernel encargado del DFA realiza esta tarea. No obstante, la implementación en GPU sigue teniendo una limitación en número de firmas paralelas que pueden ser procesadas. Este corte se asocia a las limitaciones de memoria que afectan a esta plataforma.

En la Figura 4.9 se comprueba la aceleración obtenida al aumentar el número de expresiones regulares asociadas a cada DFA. El número de firmas evaluadas por cada DFA será un compromiso entre unidades de procesamiento paralelo y exigencias de memoria. Así pues, mientras que un número limitado facilitará su localización en memorias cachés, un número elevado garantizará un menor número de accesos a memoria.

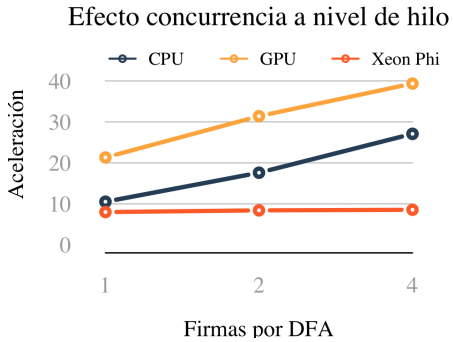


Figura 4.9. Aceleración para la configuración óptima variando las firmas por DFA.

4.2.8. Resultados FPGA

La implementación realizada en la plataforma FPGA tiene la particularidad de necesitar el mismo número de ciclos para procesar cualquier firma al limitar el payload. De este modo determinar el siguiente estado dado un byte de entrada tarda sólo un ciclo. Como la longitud del payload de los flujos está limitada a 256 caracteres, el tiempo que se tarda en procesar cada firma son 256 ciclos. Si

hay varias firmas implementadas simultáneamente, el tiempo para procesar ese número arbitrario es de 256 ciclos. Esto hace que lo más adecuado para obtener la máxima aceleración del algoritmo sea saturar los recursos disponibles en la Virtex 7 VX690T con el mayor número de firmas posibles.

Por otro lado, como la magnitud que se mantiene constante son los ciclos de reloj, es totalmente dependiente de la frecuencia de reloj seleccionada. A su vez esta está limitada por la complejidad de la electrónica que efectúa la operación de salto entre estados. En las pruebas realizadas se ha establecido una frecuencia de 125 MHz, equivalente a un periodo de 8 ns, común para configuraciones de PCIe. Esto nos lleva a un resultado de 488,28 kiloflujos por segundo, o si se aplica paralelismo de sesiones, este valor crece linealmente. Para 8 sesiones simultáneas, se alcanza el valor de 3,906 millones de flujos por segundo. Hay que tener en cuenta que estos son resultados obtenidos en el supuesto de que se procesen 145 firmas en paralelo.

En la Figura 4.10 se representa el histograma asociado al consumo de recursos por cada una de las firmas. En el eje de las abscisas se representa el número de slices utilizadas por las firmas; en el de ordenadas, el número de firmas que se localizan en ese rango. La función de distribución acumulada se superpone para indicar el porcentaje respecto del total de firmas en función del aprovechamiento de área. Esta gráfica muestra que el 80 % de las firmas ocupan 200 LUTS o menos. En la Tabla 4.3 se exponen los recursos consumidos por el diseño global. En el caso presentado cada firma ocupa en media 0,065 % de los recursos LUTs (siendo la mediana un 0,019 %), requiriendo la firma más grande 3,46 % y la más pequeña 0,0009 %.

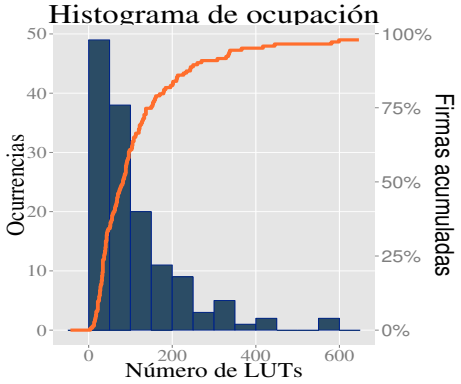


Figura 4.10. Histograma que muestra el uso de LUTs en la plataforma FPGA VX690T.

Recurso	Total	1 sesión	8 sesiones
Slices	108300	11,55 %	75,21 %
LUT Log.	433200	8,34 %	66,65 %
LUT Mem.	174200	0	0
Block RAM	1470	0	0

Tabla 4.3: Área ocupada en Virtex 7 VX690T para distinto número de sesiones paralelas.

4.2.9. Comparación de las tecnologías

Una vez localizada la configuración óptima para cada una de las arquitecturas, se procede a evaluar el rendimiento en unas pruebas los más justas posibles donde no se favorezca ninguna arquitectura. Se eliminan accesos a disco, es decir, la información se encontrará siempre en memoria principal para el caso de CPU/*Xeon Phi*, mientras que para el modelo FPGA se asume que los datos se encuentran ya en la placa dispuestos para su procesamiento. Para el caso de la GPU donde la información debe ser transferida por el equipo anfitrión, se lanzan tantas unidades de trabajo como sea posible, de modo que el pipeline del proceso siempre se encuentra ocupado solapándose cómputo con transferencia.

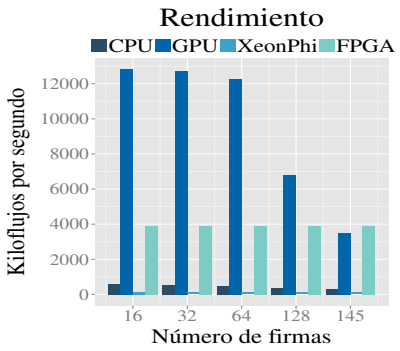


Figura 4.11. Comparación empírica usando la mejor estrategia por plataforma.

Se establece el límite de una única firma por DFA para la plataforma hardware reconfigurable donde el aumento en el número de estados provoca que no sea viable su implementación. El resto de opciones procesan 4 firmas simultáneamente. La Figura 4.11 indica el rendimiento para una traza PCAP de 50GB. Todas las plataformas mejoran la capacidad de cómputo del programa software inicial. GPU y FPGA destacan por mejorar en varios órdenes de magnitud la mejor versión alcanzada utilizando exclusivamente el diseño de CPU.

Conclusiones

A lo largo de esta sección se han evaluado distintas aproximaciones al problema. Si el número de firmas es muy limitado al igual que el número de sesiones, el coste asociado a la transferencia de los datos es realmente elevado y no procede utilizar coprocesamiento. Si el número de sesiones es mayor se puede aplicar procesamiento vectorial, mientras que si el número de firmas crece el paradigma multihilo tiene sentido. Es por ello, que dispositivos capaces de manejar grandes volúmenes de datos de manera paralela mejoran los resultados. Destacar que el rendimiento obtenido por la plataforma *Xeon Phi* ha sido inferior a las otras aproximaciones. A pesar de ser un dispositivo que ofrece unas grandes prestaciones para el procesamiento, el acceso a memoria no ha cumplido de manera satisfactoria con las exigencias del problema. FPGA y GPU han rivalizado en esta comparativa. GPU resulta idónea para un número bajo y medio de firmas. FPGA siempre que el área lo permita garantizará la ejecución simultánea de múltiples autómatas encargados de la búsqueda de coincidencias.

4.3. Seguridad de la red

La monitorización de red no solo es útil para la gestión de la misma, siendo posible determinar el buen funcionamiento de las aplicaciones, los diferentes sistemas de la red, confirmar las configuraciones de los equipos, etc. También permite obtener información útil para la seguridad, siendo posible conocer que conexiones pueden ser sospechosas, confirmar que el firewall o el IDS funcionan, etc. En este ámbito existen numerosas publicaciones y herramientas que pueden ser integradas dentro de la arquitectura propuesta. En el caso de herramientas Big Data para seguridad de red existen dos que se integran particularmente bien con el ecosistema Hadoop, Spark, etc: Apache Spot[1] y Apache Metron[2]. Ambas herramientas son similares en su objetivo: El análisis de datos de seguridad/ciberseguridad en tiempo real en plataformas Big Data. En particular, ambas aportan soluciones para integrar datos de red, y detectar, por ejemplo, conexiones sospechosas. Lo que las diferencia es que plantean aproximaciones distintas, y fundamentalmente, que al estar Apache Spot impulsada por Cloudera y Apache Metron impulsada por Hortonworks, la integración con sus respectivas distribuciones de Hadoop es

[1] `http://spot.incubator.apache.org/`
[2] `https://es.hortonworks.com/apache/metron/`

más sencilla. Por ese motivo hemos seleccionado Apache Spot para el análisis de datos de red enfocados a seguridad, ya que usamos Cloudera en nuestro clúster Hadoop.

4.3.1. Apache Spot

La propia web de Apache Spot lo describe como "un proyecto de ciberseguridad impulsado por la comunidad, construido desde cero, para aplicar análisis avanzados a todos los datos de telemetría de Tecnologías de la Información en una plataforma abierta y escalable. Es un software de código abierto para aprovechar los conocimientos del análisis de flujos y paquetes. Spot acelera la detección, investigación y resolución de amenazas a través del aprendizaje automático y consolida todos los datos de seguridad de la empresa en un centro integral de telemetría de Tecnologías de la Información basado en modelos de datos abiertos. Las capacidades de escalabilidad y aprendizaje automático de Spot dan lugar a un ecosistema de aplicaciones basadas en ML que pueden ejecutarse simultáneamente en un solo conjunto de datos enriquecido y compartido para brindar a las organizaciones la máxima flexibilidad analítica".

La instalación de Apache Spot en un clúster Cloudera es relativamente sencilla, aunque sigue siendo todavía muy manual, y presenta algunos requisitos bastante específicos. Lo mejor es disponer de un equipo o equipos adicionales dedicados a esta herramienta, y así evitar alterar la configuración del resto de equipos del clúster. Para su funcionamiento requiere un clúster Big Data con los siguientes servicios: HDFS, Hive, Impala, Kafka, Spark, Yarn y Zookeeper. Todos ellos muy habituales. Adicionalmente, Apache Spot añade tres nuevos servicios, que llama componentes: un componente de ingestión que deberá ser instalado en un equipo frontera del clúster, ya que se encarga de recibir y colectar los datos de red (flujos, paquetes, etc.), un componente de machine learning, encargado de ejecutar los diferentes modelos computacionales en los equipos de cómputo del clúster, y finalmente un componente de analítica operacional, que es básicamente un Notebook de Python modificado para funcionar como interface gráfico, que permite mostrar los datos generados por Apache Spot, e interactuar con el experto en seguridad.

Componente de ingestión Este componente necesita como fuente de datos los flujos de red y los paquetes DNS. Maneja una tercera fuente de datos que son registros de proxy, que en nuestro caso no tenemos. En cuanto los flujos de red, es

compatible con Netflow v5 y v9. En este sentido, la herramienta empleada para generar flujos, FlowProcess [6], ofrece muchos más datos de cada flujo de los generados por defecto por Netflow, por lo que es necesario realizar una etapa de pre-procesado de los flujos para eliminar aquellos campos que no son necesarios. Los paquetes de DNS, por su parte, también deben ser pre-procesados, ya que inicialmente los ficheros PCAP almacenados en HDFS incluyen todo el tráfico capturado. El pre-procesamiento consiste simplemente en usar 'tcpdump' y filtrar por puerto 53 (DNS). Aunque al ser un proyecto open source se podría modificar el código para adaptarlo a los ficheros de entrada, es preferible no realizar estas modificaciones para poder actualizar en cualquier momento la versión de Apache Spot, ya que se trata de un proyecto en su estado inicial y que cambia constantemente.

Componente de machine learning Apache Spot utiliza topic modeling para descubrir un comportamiento normal y anormal. Trata la recopilación de registros relacionados con una IP como documento y utiliza el algoritmo Latent Dirichlet Allocation (LDA) para descubrir estructuras semánticas ocultas en la recopilación de dichos documentos. LDA es un modelo bayesiano de tres niveles en el que cada palabra de un documento se genera a partir de una mezcla de un conjunto subyacente de temas [141]. El modelo LDA ejecutado asigna a cada entrada de registro de red una probabilidad estimada (puntuación). Los eventos con puntuación más baja se marcan como "sospechosos" para un análisis posterior.

Componente de visualización Apache Spot ofrece una GUI orientada a mostrar las entradas marcadas como sospechosas. Y adicionalmente ofrece un flujo que permite avanzar en el análisis de esas entradas sospechosas, llegando incluso a ofrecer editar tu propio código para realizar un análisis más a medida. Este flujo está orientado a ayudar al experto en seguridad a identificar o descartar amenazas. Por otro lado, durante la preparación de los datos para mostrar en la GUI, el componente de visualización permite usar fuentes externas (si se configuran) para ofrecer información de geolocalización, aplicar listas negras, etc. En la figura Figura 4.12 se puede ver una captura de pantalla de la GUI de Apache Spot con datos obtenidos de la red de los laboratorios de la Escuela Politécnica Superior de la UAM.

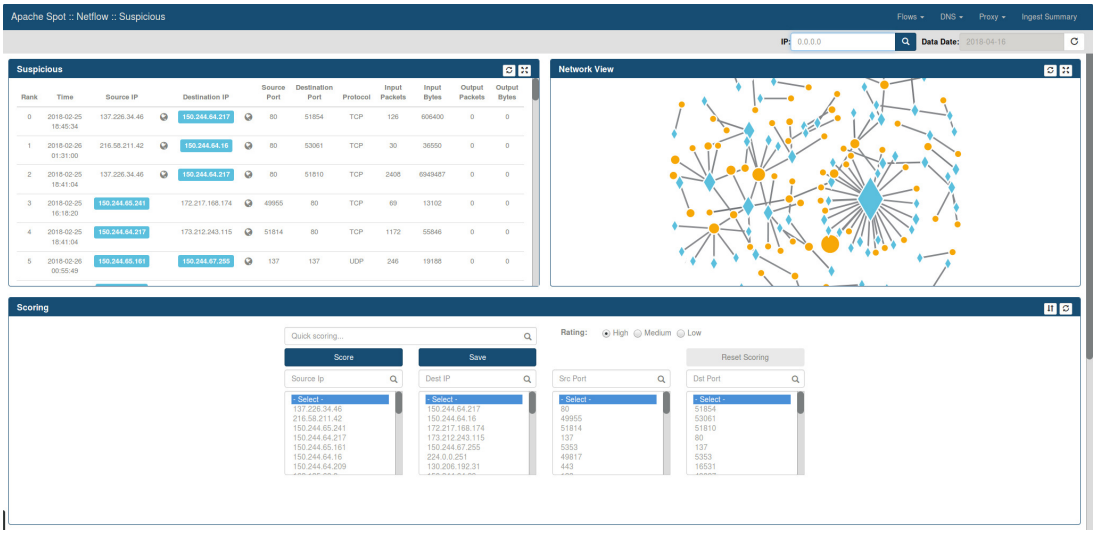


Figura 4.12. GUI de Apache Spot. Datos obtenidos de la red de la EPS.

4.3.2. Complementando Apache Spot

Apache Spot es una herramienta en desarrollo y por tanto todavía hay mucho trabajo por realizar para que se convierta en una herramienta completa. Con el objetivo de mejorar la eficacia de la herramienta, y al margen de considerar que el uso de LDA (topic modelling) y la forma de aplicarlo es muy interesante, se decide aplicar otros modelos de aprendizaje sobre los datos disponibles para complementar los resultados obtenidos por Apache Spot.

Aprendizaje no supervisado: K-Means

Una primera aproximación es proponer el uso de otro modelo de aprendizaje automático no supervisado para detectar flujos sospechosos. Al igual que en el caso de LDA, aunque existen muestras de datos que se podrían usar para entrenar modelos supervisados, en el ámbito de las redes los ataques evolucionan muy rápido, por lo que es preferible usar modelos de aprendizaje no supervisado con el objetivo de ayudar a reducir el conjunto de conexiones a analizar, de modo que aquellas conexiones marcadas como sospechosas son las que deben ser analizadas manualmente por un experto. A partir de la identificación positiva de estas conexiones sospechosas, será posible disponer de una muestra de conexiones que podría emplearse para afinar los resultados en análisis futuros, dando paso a la posibilidad de usar modelos supervisados.

Como modelo de aprendizaje no supervisado alternativo se propone uno de los más utilizados, K-means. Partiendo de los registros de flujos generados por FlowProcess, y cuyos campos se muestran en la Tabla 4.4, se quiere agrupar los flujos en diferentes clústeres con similares características, y obtener los centroides de dichos clústeres. Aquellas conexiones que se encuentren muy alejadas del centroide correspondiente al clúster al que pertenece, se puede considerar anómala. Al igual que en el caso de LDA en Apache Spot, la distancia se puede considerar como la puntuación de la conexión. En este caso, una mayor puntuación (distancia) indicará que la conexión es sospechosa.

Campo	Descripción
flowid	Identificador de flujo
src	Dirección IP origen
dst	Dirección IP destino
mac_src	MAC origen
mac_dst	MAC origen
proto	Protocolo
sport	Puerto origen
dport	Puerto destino
pkts	Número de paquetes
bytes	Total de bytes
first	Tiempo del primer paquete
latest	Tiempo del último paquete
duration	Duración del flujo en segundos
thr_packpers	Paquetes por segundo
thr_bytespers	Bytes por segundo
max_pkt_size	Tamaño máximo de paquete
min_pkt_size	Tamaño mínimo de paquete
avg_pkt_size	Tamaño medio de paquete
std_pkt_size	Desviación estándar del tamaño de paquete
max_int_time	Tiempo máximo entre paquetes
min_int_time	Tiempo mínimo entre paquetes
avg_int_time	Tiempo medio entre paquetes
std_int_time	Desviación estándar de tiempo entre paquetes
syn_rtt	RTT
fins, syns, resets, pushs, acks, urgs, cwrs, eces	Banderas de TCP

Tabla 4.4: Campos de datos para flujos generados por FlowProcess

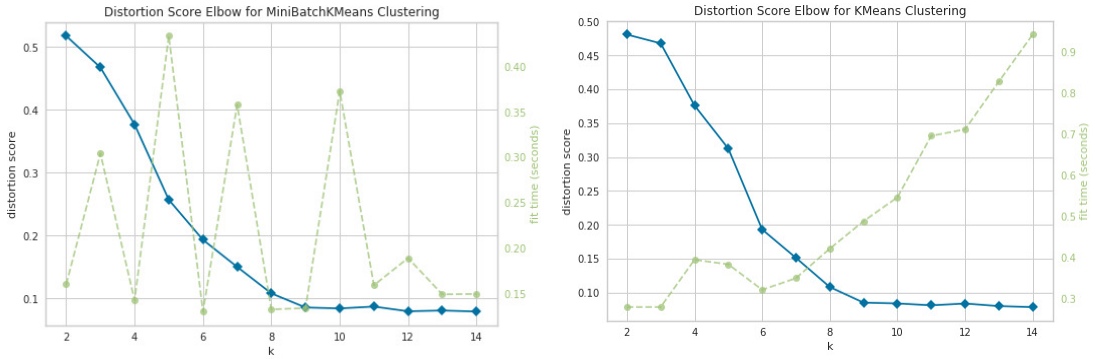
Partiendo de un conjunto de flujos de red obtenidos de la red de los laboratorios de la Escuela Politécnica Superior de la UAM, se representan en la figura Figura 4.13, todos los campos con datos numéricos. Los campos 'flowid', 'src', 'dst', 'mac_src', 'mac_dst', 'first', y 'latest' incluyen datos categóricos y se eliminan ya que en K-Means pueden inducir a errores.

Figura 4.13. Análisis de los campos obtenidos de FlowProcess (excepto valores categóricos)

Con los campos restantes ya se está en disposición de ejecutar K-Means. El primer paso es determinar el valor óptimo de K. Una de las métricas que se usa comúnmente para comparar los resultados en diferentes valores de K es la distancia media entre los puntos de las muestras y el centroide de grupo al que pertenece. Dado que aumentar el número de clústeres siempre reducirá la distancia a los puntos de las muestras, al aumentar K siempre disminuirá esta métrica, hasta el extremo de llegar a cero cuando K es igual que la cantidad de puntos de muestras. Por lo tanto, esta métrica no se puede usar como único objetivo. En cambio, se traza la distancia media al centroide como una función de K y el "punto del codo",

donde la velocidad de disminución cambia bruscamente, se puede usar para determinar aproximadamente K. Esta técnica se conoce como método del codo (elbow method).

Para ejecutar el método del codo vamos a emplear una muestra aleatoria de 50000 flujos. Se utilizan tanto el algoritmo K-Means como MiniBatch K-Means con el objetivo de validar que ambas aproximaciones dan el mismo valor para K. Elegimos un rango amplio, desde K = 2 hasta K = 14. Los resultados se muestran en Figura 4.14a y Figura 4.14b. Adicionalmente, se muestra también el tiempo que se ha empleado para ajustar las funciones, lo que permite comparar cada uno de los algoritmos (K-Means y MiniBatch) para los diferentes valores de K.

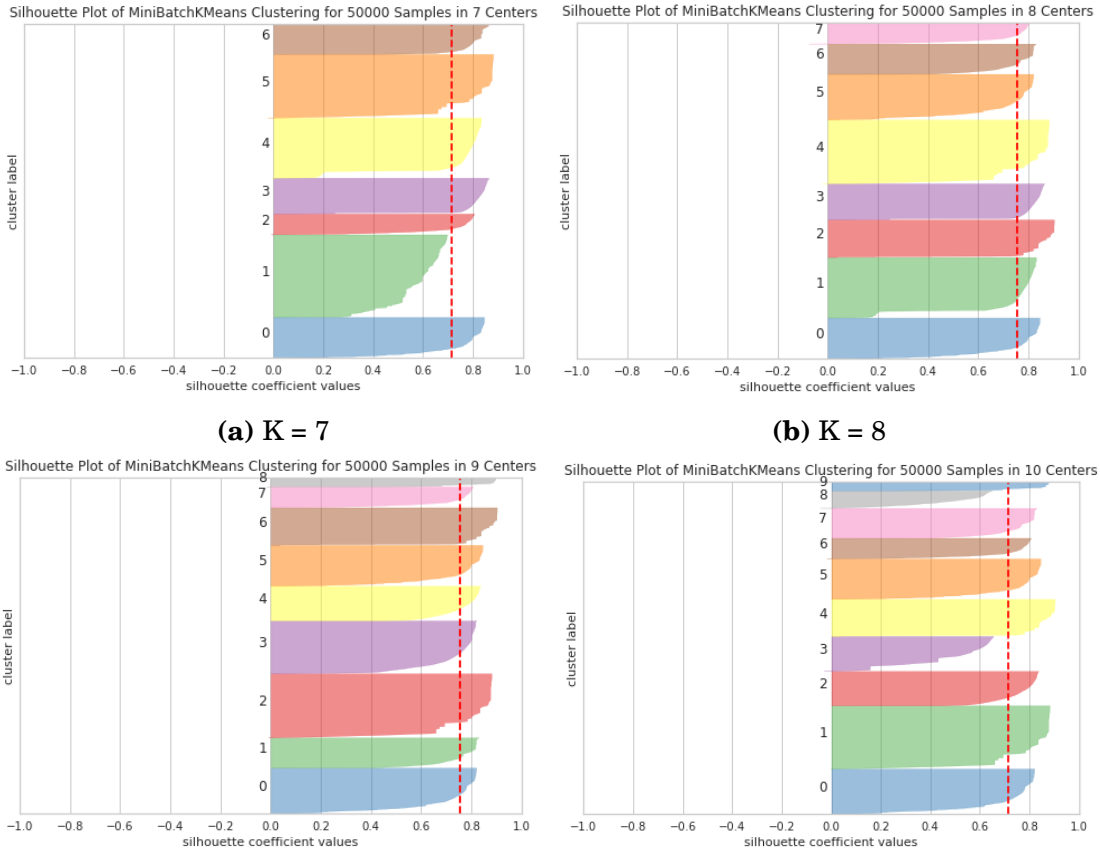


(a) MiniBatch K-Means **(b)** K-Means

Figura 4.14. Resultado de aplicar la técnica del codo

Como puede apreciarse, el método del codo parece funcionar razonablemente bien. En ambas gráficas se ve una curva suave, particularmente en el caso de MiniBatch K-Means, pero no queda claro dónde está el codo. Todo apunta a K = 8 o K = 9. En casos como éste, se puede probar un método diferente para determinar K, como puede ser Silhouette. El análisis Silhouette se usa para estudiar la distancia de separación entre los clústeres resultantes. El gráfico de Silhouette muestra una medida de como de cerca está cada punto de las muestras del resto de puntos de los clústeres vecinos y, por lo tanto, nos permite evaluar el número de clústeres visualmente. Esta medida tiene un rango de [-1, 1]. Si la medida está cerca de +1 indica que la muestra está lejos de los clústeres vecinos, por lo que se encuentra bien clasificada. Un valor de 0 indica que la muestra está en el límite de decisión entre dos clústeres vecinos, y el valor negativo indica que la muestra podría haberse asignado al clúster incorrecto.

En la Figura 4.15a, Figura 4.15b, Figura 4.15c y Figura 4.15d se muestran los resultados de Silhouette para K = [7, 8, 9, 10]. El eje X representa los coeficientes de Silhouette, que es como se conoce a las medidas antes mencionadas. La línea vertical roja de las figuras indica el coeficiente medio para toda la muestra. En este caso, parece que K = 8 y K = 9 están muy cercanos a 0.8, mientras que K = 7 a K = 10 presentan un coeficiente menor. Dado que K = 8 y K = 9 están muy parejas, elegimos K = 8 por tener menos clústeres.



(a) K = 7 **(b)** K = 8

(c) K = 9 **(d)** K = 10

Figura 4.15. Resultado de aplicar Silhouette

Una vez que se ha encontrado el número de clústeres, procedemos a ejecutar el modelo K-Means para ese número K. La Figura 4.16 muestra los clústeres resultantes, así como los centroides de cada clúster. Como se ha indicado al principio de esta sección, se asume que las conexiones anómalas son aquellas que se encuentran muy alejadas del centroide del clúster al que pertenece. Es evidente en la Figura 4.16, que, por ejemplo, los clústeres 2 y 0 tienen puntos muy alejados del centroide, por tanto, son candidatos a ser clasificadas como conexiones anómalas.

El paso siguiente es ordenar las muestras (flujos) por distancia de su centroide, de mayor a menor, y seleccionar aquellas que superan una distancia determinada. Posteriormente, se cruzan los resultados de K-Means y LDA y aquellos casos donde coinciden, se les da mayor prioridad en la representación en Apache Spot. Para ello, es necesario realizar modificaciones en la tabla Hive/Impala donde Apache Spot almacena las conexiones sospechosas.

Finalmente, como el tiempo de ejecución de K-Means es significativamente alto, ya que se usa un script en Python con la librería scikit-learn, se propone una solución más orientada a "tiempo real". La idea es ejecutar por la noche K-Means con el conjunto de flujos capturados durante el día, de modo que se obtengan los centroides de cada uno de los 8 clústeres. Posteriormente, al día siguiente, a partir de grupos reducidos de flujos recién generados, se determina a qué clúster pertenecen y se calcula la distancia a su centroide. Si la distancia supera el valor estimado, se marcan como anómalos. Este sistema permite clasificar los flujos durante el día "casi en tiempo real", y se aprovecha la noche, donde hay menos tráfico, para "entrenar" el modelo.

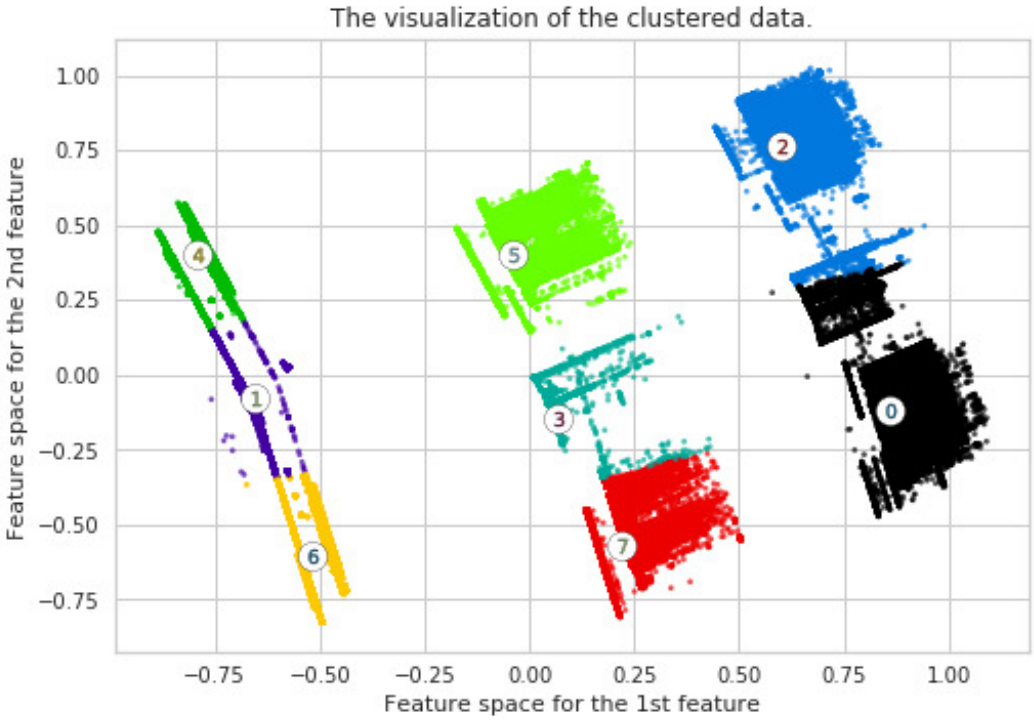


Figura 4.16. Clusterizado con K-means para K = 8

Clasificación de flujos: Redes Neuronales Profundas

Con el objetivo de seguir añadiendo modelos de aprendizaje automático a la instalación de Apache Spot, se propone el desarrollo de un clasificador de flujos basado en redes neuronales profundas. El objetivo es comprobar si es posible, a partir de los diferentes campos que se obtienen de los flujos de red en FlowProcess, determinar el tipo de aplicación que ha generado el flujo. Para ello, se parte de la misma muestra de datos empleada en el algoritmo K-Means, esto es, datos obtenidos de la red de los laboratorios de la Escuela Politécnica Superior de la UAM.

Al igual que en el caso de K-Means, se descartan los valores categóricos, además de los flags. También se eliminan los campos 'sport', 'dport', 'std_pkt_size', 'std_int_time' y 'syn_rtt'. Los dos primeros se descartan porque una de las formas de determinar el tipo de aplicación que ha generado el flujo es el valor del puerto destino, que a su vez también se corresponde con el puerto origen en las respuestas. El resto de campos se eliminan porque tras varias pruebas se determina que no aportan mejores resultados, y de este modo, se reduce el tamaño de datos a emplear durante el entrenamiento / validación.

Dado que vamos a implementar un clasificador, que es un modelo de aprendizaje supervisado, necesitamos una etiqueta para clasificar el conjunto de datos de entrenamiento. Para etiquetar los flujos, vamos usar el mínimo entre el puerto origen (sport) y el puerto destino (dport) de cada flujo. Y, además, solo se selecciona el flujo si ese valor mínimo es menor o igual a 1024. Los servicios más conocidos y habituales utilizan puertos menores de 1024. Además, sabemos que el tráfico de muestra se obtiene detrás de un firewall que solo permite ciertos servicios bien conocidos como HTTP/HTTPS, SSH, etc.

Adicionalmente, sobre los registros de flujo se aplican los siguientes filtros:

- El valor de 'dport' o 'sport' debe ser menor de 1024.

- El número de paquetes del flujo debe ser mayor que 1 y menor que 1000, a excepción de los flujos de DNS (puerto 53) donde se permiten flujos de un único paquete. El número máximo de 1000 paquetes se aplica para no distorsionar la fase de entrenamiento.

- Tras aplicar la etiqueta a cada flujo, se eliminan todos los registros cuya etiqueta no aparezca al menos 10000 veces en el fichero. Se trata de eliminar

aquellos flujos que no tengan un volumen suficiente para que el clasificador aprenda correctamente.

Tras aplicar el filtrado y etiquetado a un fichero de pruebas con 5 millones de flujos nos quedan 2.973.279 flujos, con las siguientes etiquetas:

Puerto	Servicio	Etiqueta
0	Sin uso. Paquetes ICMP.	0
22	SSH	1
53	DNS	2
67	BOOTP	3
80	HTTP	4
123	NTP	5
137	Servicios de nombres NETBIOS	6
138	Servicios de datagramas NETBIOS	7
443	HTTPS	8
445	SMB	9
993	IMAP	10

La implementación del clasificador se realiza empleando TensorFlow, y particularmente un estimador DNNClassifier con 3 capas ocultas de 100 neuronas cada una, un ratio de aprendizaje de 0.08, 40.000 pasos, y un tamaño de bloque de 150. Estos valores se han determinado tras realizar varias pruebas para determinar la mejor configuración. Como optimizador se usa Adagrad. Como paso previo antes de aplicar el clasificador, es necesario normalizar los datos por el método de máximos y mínimos, y se divide el conjunto de datos en un 70% para entrenamiento y el 30% restante para validación. El resultado de la fase de entrenamiento se puede ver en la Figura 4.17. Tras finalizar el entrenamiento se obtiene una precisión del 92%. La matriz de confusión resultante puede verse en la Figura 4.18.

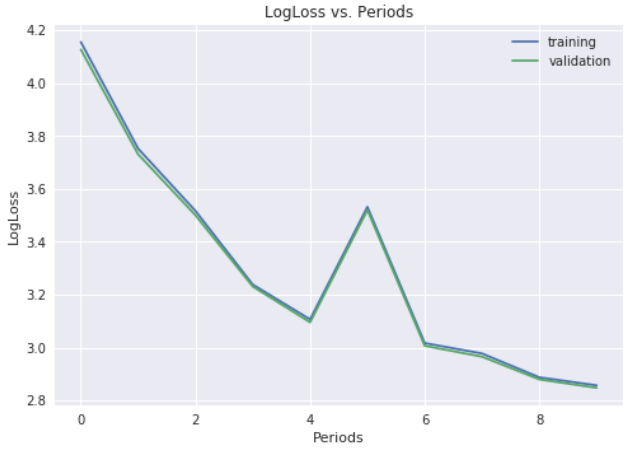


Figura 4.17. Resultados del entrenamiento

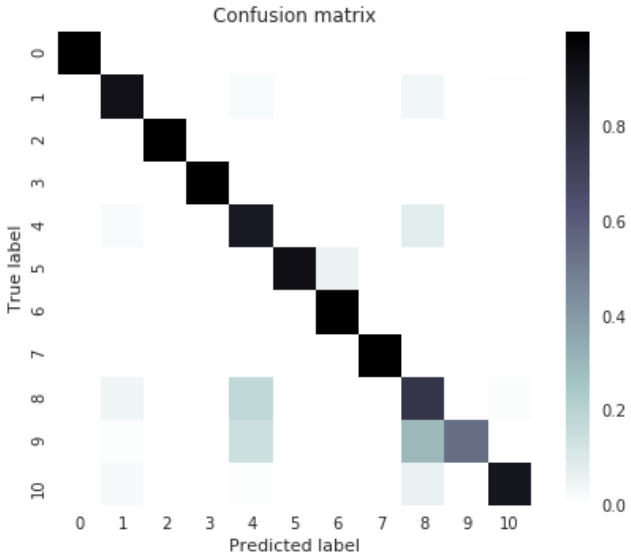


Figura 4.18. Matriz de confusión sobre el conjunto de datos de validación

Analizando la matriz de confusión se observa que el clasificador confunde algunos servicios, motivo por el cual la precisión del modelo no es más elevada. Por ejemplo, es lógico confundir HTTP y HTTPS ya que ambos pueden corresponder con navegación web, aunque uno de ellos esté cifrado. De igual modo, SSH y HTTPS se pueden confundir si en ambos casos se produce una descarga de ficheros. Lo mismo pasa en el caso de HTTP y SMB en las descargas de ficheros. Por lo tanto, existen al menos 4 servicios que pueden a priori confundirse en este escenario.

Una vez entrenado el clasificador, se procede a verificar su funcionamiento utilizando un nuevo fichero con 10 millones de flujos distintos (corresponden a otro día). Antes de clasificarlos, se realizan las transformaciones necesarias sobre los datos. Tras aplicar el clasificador, la precisión obtenida es del 85 %, y la matriz de confusión se muestra en la Figura 4.19. Al igual que en el entrenamiento, la matriz de confusión muestra que se confunden los mismos servicios. La caída en la precisión puede deberse a la variabilidad del tráfico en esta muestra con respecto a la de entrenamiento, ya que el tráfico de los laboratorios no es el mismo todos los días, ni todas las horas. Los resultados del clasificador se emplean, al igual que en el caso de DPI, para identificar el tipo de servicio del flujo con mayor detalle. Son 2 técnicas con el mismo objetivo, aunque diferente aproximación.

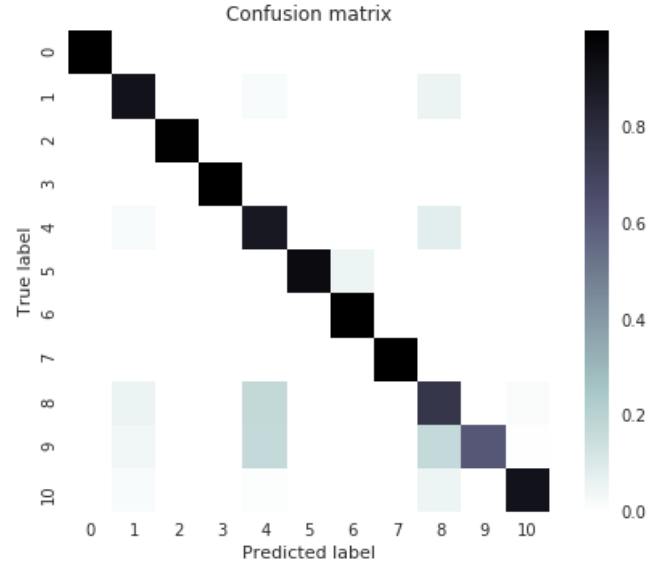
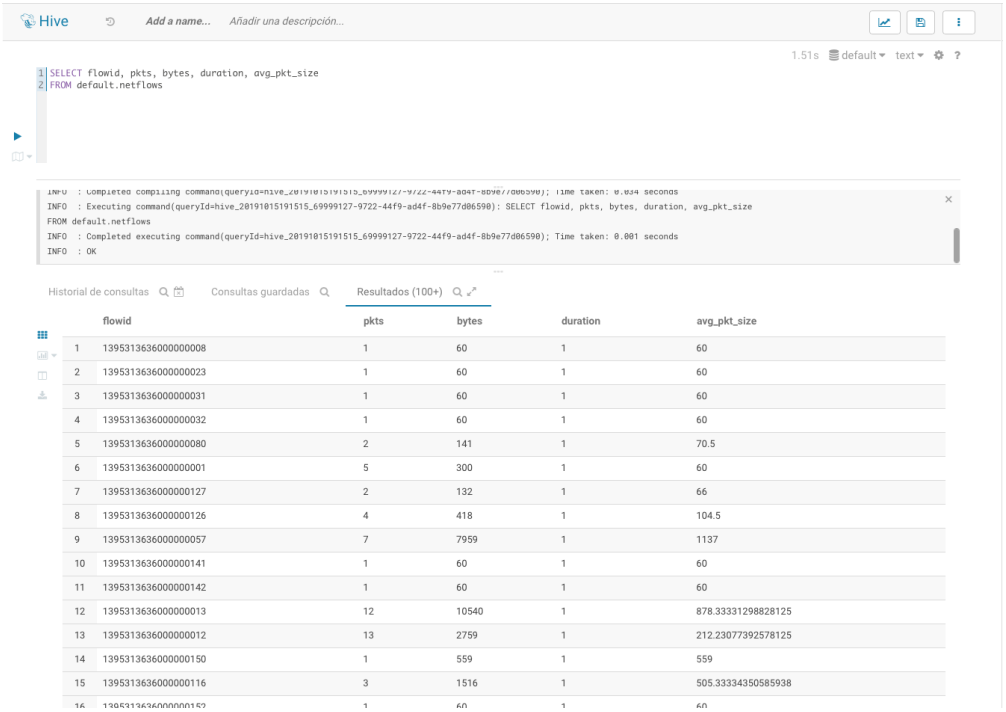


Figura 4.19. Matriz de confusión para el fichero de 10 millones de flujos

4.4. Visualización

El último paso necesario para obtener un sistema de monitorización completo es la visualización de la información. El componente de visualización ideal debe mostrar los sucesos más relevantes en el menor espacio posible, así como la información lo más agregada y compacta posible. De esta forma y con un tiempo mínimo, es posible, en términos generales, detectar si la red está funcionando como se esperaba o presenta algún tipo de anomalía evidente. Así mismo, este componente debe permitir profundizar e ir obteniendo información más concreta a petición

del usuario, hasta, si fuese preciso, con la menor granularidad posible, lo que en nuestro caso se traduciría en mostrar paquetes de red por pantalla. La necesidad de tener una representación gráfica multi-granular viene impuesta por el problema del volumen de información que maneja el sistema. Si bien representar todos los datos en crudo, así como la salida completa de cada una de las herramientas de análisis y predicción, representa una tarea computacional elevada, también representa un problema humano a la hora de discernir la información relevante para un momento dado. Este componente, a pesar de su complejidad, suele ser comúnmente infravalorado y subestimado en muchas situaciones, cuando en algunas ocasiones puede llegar ser un componente tanto o más complejo que el resto del sistema en el que se integra.



	flowid	pkts	bytes	duration	avg_pkt_size
1	1395313636000000008	1	60	1	60
2	1395313636000000023	1	60	1	60
3	1395313636000000031	1	60	1	60
4	1395313636000000032	1	60	1	60
5	1395313636000000080	2	141	1	70.5
6	1395313636000000001	5	300	1	60
7	1395313636000000127	2	132	1	66
8	1395313636000000126	4	418	1	104.5
9	1395313636000000057	7	7959	1	1137
10	1395313636000000141	1	60	1	60
11	1395313636000000142	1	60	1	60
12	1395313636000000013	12	10540	1	878.33331298828125
13	1395313636000000012	13	2759	1	212.23077392578125
14	1395313636000000150	1	559	1	559
15	1395313636000000116	3	1516	1	505.33334350585938

Figura 4.20. Ejemplo de búsqueda en Hive

No obstante, los sistemas de visualización son un problema recurrente y por ello existen multitud de herramientas de visualización distintas. Dentro de este capítulo se han mencionado dos posibles formas de visualización de datos, que, a su vez, permiten realizar análisis integrado. En este caso, hablamos de Apache Hive (ver Sección 4.1) y Apache Spot (ver Subsección 4.3.1). En el caso de Apache Hive, se provee de una interfaz tipo SQL que permite gran flexibilidad a la hora de mostrar tablas y gráficos sencillos exactamente como el usuario quiere en un

momento dado. En la Figura 4.20 se muestra una consulta simple en Hive, similar a las empleadas en la Sección 4.1 para obtener datos de flujos de red. Sin embargo, para poder realizar este tipo de consultas se requieren conocimientos básicos de programación, así como conocer la estructura de almacenamiento de los datos en el sistema, lo que limita el alcance y facilidad de uso.

Por otro lado, Apache Spot provee una interfaz muy visual, que permite al usuario interactuar con la información a diferentes niveles. Además, incluye la posibilidad de identificar aquella información que requiere un análisis más complejo, llegando incluso a permitir añadir código para manipular los datos. Por tanto, es una herramienta muy potente, pero orientada a expertos en seguridad con conocimientos de programación.

Al igual que los dos ejemplos de sistemas de visualización mostrados, muchas de las soluciones de visualización en el ámbito de las redes de comunicaciones, requieren conocimientos previos para permitir un uso efectivo de las mismas. Es por ello que en la actualidad, una de las opciones más recurrentes para resolver los problemas de visualización son las interfaces basadas en *Dashboard*s. Incluso, algunas aplicaciones de monitorización comerciales con más renombre basan sus esfuerzos en modificaciones de algunos de estos sistemas [142]. Dentro de estos sistemas destaca fundamentalmente Grafana [143], como una de las opciones *Open Source* más usadas, seguida de cerca por Kibana [144]. La diferencia fundamental entre ambas es el objetivo del desarrollo. Mientras que Grafana se enfoca en múltiples y heterogéneas fuentes de datos, Kibana se centra en mejorar y añadir formas de representar los datos almacenados en Elasticsearch. Un ejemplo de *Dashboard* para visualización de datos de red, creado con Grafana, se muestra en la Figura 4.21. Los datos mostrados son reales, y corresponden con equipos en producción, es por ello que se han omitido algunos detalles.

El concepto de *Dashboard* parte de una interfaz gráfica basada en grupos de paneles en donde cada uno de ellos muestra una información concreta. Estos paneles pueden tener multitud de formas: series temporales, mapas de calor, tartas, tops, etc. A su vez, estos *Dashboards* suelen acotar la información que se está mostrando a un intervalo temporal. Además, gracias a variables o plugins a medida, es posible dotar a estos *Dashboards* de funcionalidades extra a medida, cómo expresiones complejas para realizar distintos tipos de filtrado, botones que efectúan operaciones sobre los datos, o incluso crear nuevos *Dashboards*. A su vez, es posible crear nuevas series temporales sobre las series ya creadas, permitiendo usarlas como límites superiores o inferiores. Estas nuevas series temporal se crean

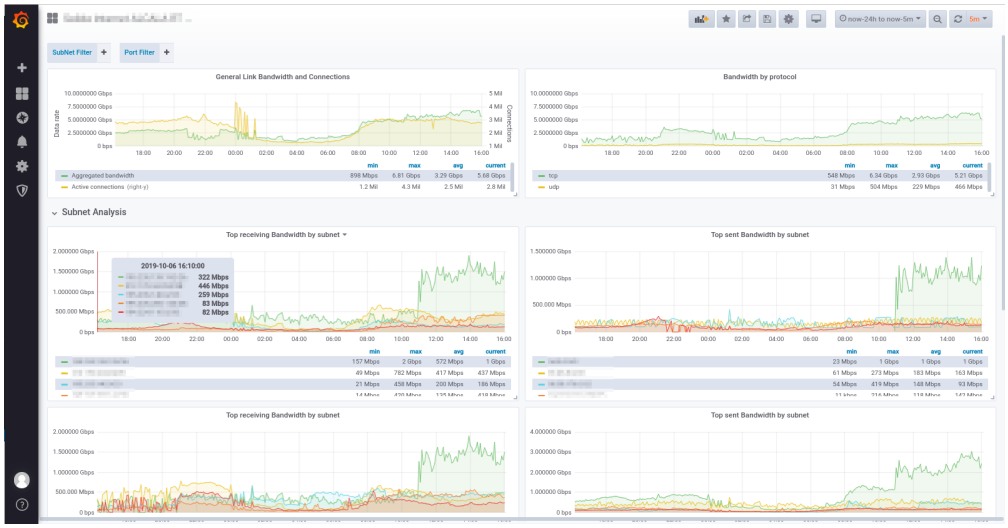


Figura 4.21. Ejemplo de *Dashboard* con Grafana

empleando algún tipo de fórmula, que a su vez permite implementar alarmas, de tal forma que, si consideramos que rebasar dichas series temporales creadas por nosotros supone una anomalía, podemos avisar de ello de manera gráfica en el propio *Dashboards*, y actuar en consecuencia. Algo muy útil en la monitorización de redes de comunicaciones.

La complejidad que de por sí tiene el desarrollo de interfaces de usuario, su usabilidad, los diferentes tipos de representaciones gráficas, etc. está fuera del alcance de esta tesis, si bien en esta sección se ha mostrado y desarrollado algunos ejemplos, con el objetivo de completar el sistema de monitorización integral planteado como objetivo de la tesis. Cabe destacar, que consideramos que las herramientas basadas en *Dashboard*s son lo suficientemente flexibles y capaces cómo para alcanzar los objetivos de visualización deseados.

4.5. Conclusiones

En este capítulo se han mostrado diferentes análisis complejos sobre los datos de red, como la detección de anomalías y problemas de seguridad en redes de comunicaciones. No obstante, aunque se trata de resultados preliminares, se abren nuevas líneas de investigación, como el uso de Machine Learning en redes de comunicaciones, que sería interesante explorar como trabajo futuro. Por ejemplo, el uso de redes neuronales y redes neuronales convolucionales profundas en la iden-

tificación tanto de anomalías y problemas de seguridad, como en la identificación y caracterización de flujos.

Por otro lado, los sistemas de visualización cómo Apache Hue (GUI de Hive), Apache Spot o Grafana, así como las distintas alternativas basadas en *Dashboards*, se muestran como una alternativa interesante para la visualización de datos de red. Sin embargo, se ha profundizado poco en este apartado, al considerar que está más relacionado con la investigación centrada en las interacciones del usuario final y en la representación de datos. Sin embargo, son una pieza fundamental de cualquier sistema de monitorización de red.

5

Conclusiones y trabajo futuro

DIVIDE entre computadoras y vencerás es uno de los lemas con los que podría definirse la filosofía y el aprendizaje obtenido en el desarrollo de esta tesis. Este lema parte de un problema recurrente en la monitorización de red: la constante reingeniería de herramientas y subsistemas con el progreso y avance tecnológico. La única solución a este gran problema es su división en pequeños problemas más simples y desligados lo máximo posible de la tecnología subyacente, distribuyendo el cómputo entre distintos equipos. No obstante, para llegar a esta gran conclusión se han obtenido resultados y conclusiones en distintos ámbitos, todos ellos necesarios para formar el sistema de monitorización distribuido objetivo de esta tesis. De forma breve las aportaciones de esta tesis han sido las siguientes:

- Obtención del estado de la red mediante sistemas activos.
- Infraestructura de captura de paquetes de red a alta velocidad (+100Gbit/s)
- Virtualización de sistemas de captura a alta velocidad
- Infraestructura para la distribución y análisis de paquetes
- Análisis de paquetes offline mediante técnicas innovadoras

5.1. Retos, contribuciones y conclusiones

Esta tesis plantea la construcción de una arquitectura de monitorización integral que incluya desde distintos sistemas de captura de datos de forma activa o pasiva, su redistribución dentro de una arquitectura paralela, su análisis y finalmente su representación gráfica (ver Figura 1.2). Esta arquitectura fue llevada a un congreso de primer nivel y mostrada como póster [145], obteniendo un elevado interés y *feedback* por parte de los académicos que asistieron, incluso, propuestas de trabajo para llevar a cabo en un entorno más industrial. En la siguiente sección se describen los retos, contribuciones y conclusiones de cada uno de los 4 pasos en la monitorización.

5.1.1. Captura de datos y métricas

El proceso de captura de datos está recogido enteramente en el Capítulo 2. Este capítulo comienza con la problemática de la monitorización activa y cómo la obtención de medidas de alta calidad y precisión solo pueden ser obtenidas mediante finas mediciones y estadística. Esta sección, se inició cómo un estudio de viabilidad de las limitaciones del software con serias dudas acerca del alcance y precisión posibles. Los resultados obtenidos fueron muy superiores a lo esperado permitiendo una publicación [18] en una revista Q2 en el momento del envío. Dicha revista otorgó un premio económico y de reconocimiento a dicho artículo a mejor artículo del año en dicha revista. Este premio fue otorgado tras el voto de los editores que forman dicha revista.

En la Sección 2.1 se recurrieron a las lecciones y conocimientos aprendidos durante la carrera y el máster para explorar nuevas opciones de captura que permitiesen alcanzar tasas cada vez mayores. Partiendo del exitoso driver HPCAP [7] se colaboró en el desarrollo de una alternativa virtualizada ya que la mayoría de sistemas actuales se ejecutan en entornos virtualizados y de Cloud. Dicha colaboración acabó en la siguiente publicación [63]. Tras obtener gran experiencia en el desarrollo y puesta a punto del driver HPCAPvf se colaboró en el desarrollo de la nueva generación del driver HPCAP40 destinado a interfaces de 40 Gbit/s y liderando el desarrollo de su alternativa virtualizada. Ambos drivers, tanto para tarjetas físicas cómo sus correspondientes funciones virtuales llevaron a la publicación de dos artículos [75, 146].

El problema de los sistemas de almacenamiento aparece ya desde las primeras versiones de HPCAP [7], en donde el almacenamiento a 10Gbit/s presentaba un reto tan grande como obtener los paquetes de la interfaz de red sin pérdidas. Este reto se incrementó al avanzar a 40 Gbit/s, momento en el cual hubo que explorar alternativas de almacenamiento (discos NVMe), así como nuevas interfaces de escritura en disco (SPDK) y tipos y métodos de construcción de RAIDs.

En la Sección 3.1 se plantea uno de los sistemas de monitorización pasiva más clásicos: la recolección de logs de sistemas. Este trabajo fue liderado y mayormente desarrollado por Carlos Vega [95], no obstante, el diseño de la arquitectura y las pruebas de rendimiento fueron realizadas a partes iguales por los autores del artículo [94]. Tanto el diseño realizado, como las lecciones aprendidas durante el desarrollo fueron la semilla perfecta para el diseño y creación del futuro Wormhole.

5.1.2. Redistribución de los datos

La arquitectura de los drivers tipo HPCAP se encontraba muy ligada a ciertos modelos de tarjetas y fabricantes, limitando en gran medida un posible uso por terceros. Por este motivo se decidió dar un paso adelante e intentar crear un sistema de captura similar a los drivers de HPCAP utilizando DPDK. Los buenos resultados obtenidos, han permitido poner en producción, en un cliente de Naudit HPCN S.L., un sistema de monitorización basado en DPDK.

Al avanzar hacia los 100 Gbit/s, a pesar de que el driver de DPDK soportaba esta tasa, ningún algoritmo de análisis lo soportaba. Incluso un almacenamiento capaz de soportar esa tasa se llenaba en pocos minutos (>15 minutos). Por este motivo se ideo un sistema de captura que fielmente integrado con un sistema de distribución de datos basado en Big Data. La idea de este sistema fue publicada inicialmente en un congreso Español con la idea de obtener feedback de nuestra idea [147]. Tras evaluar el estado del arte y quedar patente la incapacidad del resto de sistemas de Streaming para alcanzar la tasa necesaria, se creó el sistema *Wormhole*, el cual fue publicado finalmente en [96]. Aunque Wormhole supera en todos los aspectos de rendimiento al resto de sistemas de Streaming, es una herramienta con muchas carencias de gestionabilidad que requieren un intenso trabajo y desarrollo futuro para poder ser usado ampliamente por la industria.

5.1.3. Análisis y visualización de los datos

Analizar los paquetes de red y llevar a conclusiones es una tarea compleja y es un área de investigación por sí misma. Uno de los primeros pasos a la hora de obtener información de los paquetes es su agrupación en el concepto de flujos [27]. Dicho resumen se ha realizado utilizando distintas herramientas ya construidas por el grupo como el Flowprocess [6] o el Detectpro [148]. En la Sección 4.2, se desarrolla un método utilizando coprocesadores de distintos tipos para inferir el contenido a nivel de aplicación de los flujos. Un proceso bastante complejo en diversos aspectos.

A través de múltiples instancias de Wormhole conectadas a analizadores de paquetes (Flowprocess o Detectpro), es posible reconstruir flujos en distintas máquinas y procesar y analizar un mayor número de ellos de forma paralela. La forma más sencilla de lograrlo es guardar paralelamente en un sistema de ficheros distribuido (por ejemplo, HDFS) y recurrir a herramientas del estado del arte para su análisis de seguridad y representación gráfica. Gracias a las múltiples librerías, es posible incluso utilizar técnicas de Machine-Learning sobre los datos con facilidad y obtener predicciones del estado de la red, detectar anomalías, etc. Dado la complejidad de esta área, la contribución en estos aspectos ha sido limitada y se pretende seguirla explorando en un futuro.

5.2. Listado formal de publicaciones

5.2.1. Artículos en revistas indexadas en JCR

- Rafael Leira, Javier Aracil, Jorge Enrique López de Vergara, Paula Roquero, Iván González. High-speed optical networks latency measurements in the microsecond timescale with software-based traffic injection. Optical Switching and Networking (2018) Vol. 29, Páginas 39–45. [18].
  Índice de impacto JCR: 1.865. **Ranking JCR**: 41/89 (Q2).
  Categoría JCR: Telecommunications (2016).
  SCIMAGO: Q2 en “Computer Networks and Communications” 2017
  Relación con la tesis: Forma parte del Capítulo 2.
  Premio: 2018 Fabio Neri Best Paper Award [149]

- Rafael Leira, Guillermo Julián-Moreno, Iván González, Francisco Javier Gómez Arribas, Jorge Enrique López de Vergara. On the performance assess-

ment of 40 Gbit/s off-the-shelf network cards for virtual network probes in 5G networks. Computer Networks Vol. 152, Páginas 133–143. [146].
Índice de impacto JCR: 2.516. **Ranking JCR**: 13/52 (Q1).
Categoría JCR: Computer Science, Hardware & Architecture (2016).
SCIMAGO: Q2 en "Computer Networks and Communications" 2017
Relación con la tesis: Forma parte del Capítulo 2.

- Carlos Vega, Paula Roquero, Rafael Leira, Iván González, Javier Aracil. Loginson: a transform and load system for very large-scale log analysis in large IT infrastructures. The Journal of Supercomputing (2017) Vol. 73, número 9, Páginas 3879–3900. [94].
Índice de impacto JCR: 1.326. **Ranking JCR**: 52/104 (Q2).
Categoría JCR: Computer Science, Theory & Methods (2016).
SCIMAGO: Q2 en "Hardware and Architecture" 2017
Relación con la tesis: Forma parte del Capítulo 3.

5.2.2. Artículos presentados en congresos

Internacionales

- Rafael Leira, Iván González, Lluis Gifre, Jorge Enrique López de Vergara. Wormhole: A 100 Gbit/s streaming engine for Big Data network monitoring systems. 2019 22nd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN) [96]
Ranking de Conferencia: B **Evaluador**: GII-GRIN-SCIE May-2018 [150]
Ranking en revista equivalente [151]: –.
Relación con la tesis: Forma parte del Capítulo 3.

- Guillermo Julián Moreno, Rafael Leira, Jorge Enrique López de Vergara, Francisco Javier Gómez Arribas, Iván González. On the feasibility of 40 gbps network data capture and retention with general purpose hardware Proceedings of the 33rd Annual ACM Symposium on Applied Computing (2018). Páginas 970-978. [75].
Ranking de Conferencia: A- **Evaluador**: GII-GRIN-SCIE May-2018 [150]
Ranking en revista equivalente [151]: Q3/4.
Relación con la tesis: Forma parte del Capítulo 2.

- Rafael Leira, Iván González, Lluis Gifre, Jorge Enrique López de Vergara Network Analysis on Lambda Architecture. ACM Internet Measurement Conference (2017). Poster session. [145].
  Ranking de Conferencia: A+ **Evaluador**: GII-GRIN-SCIE May-2018 [150]
  Ranking en revista equivalente [151]: Q1/2.
  Relación con la tesis: Forma parte del Capítulo 3.

- Victor Moreno, Rafael Leira, Iván González, Francisco Javier Gómez Arribas. Towards High-Performance Network Processing in Virtualized Environments. International Conference on High Performance Computing and Communications (HPCC) (2015) [63].
  Ranking de Conferencia: Core: B ; Microsoft Academic: C
  Relación con la tesis: Forma parte del Capítulo 2.

Nacionales

- Rafael Leira, Paula Roquero, Carlos Vega, Iván González, Javier Aracil. Hpsengine: Motor de alto rendimiento y baja latencia para el procesamiento distribuido en tiempo real XXVI Jornadas de Paralelismo, Salamanca, Spain, Sep 2016. [147].
  Ranking de Conferencia: Sin Ranking
  Relación con la tesis: Forma parte del Capítulo 3.

- José Fernando Zaco, Raúl Martín, Rafael Leira, Iván González, Gustavo Sutter, Francisco Javier Gómez Arribas. Clasificación de tráfico de red mediante aceleradores hardware. XV Jornadas de Computación Reconfigurable y Aplicaciones 2015, Córdoba, Spain, Sep 2015 [152].
  Ranking de Conferencia: Sin Ranking
  Relación con la tesis: Forma parte del Capítulo 4.

- Ruben García-Valcárcel, Rafael Leira, Iván González, Jorge Enrique López de Vergara. Monitorización y análisis de tráfico de red con apache hadoop. Jornadas de Ingeniería Telemática (JITEL), Palma de Mallorca, Spain, Oct 2015. [153].
  Ranking de Conferencia: Sin Ranking
  Relación con la tesis: Forma parte del Capítulo 4.

- Ruben García-Valcárcel, Rafael Leira, Iván González, Francisco Javier Gómez Arribas. Evaluando apache hadoop para análisis de tráfico de red XXV

Jornadas de Paralelismo, Córdoba, Spain, Sep 2015. [154].
Ranking de Conferencia: Sin Ranking
Relación con la tesis: Forma parte del Capítulo 4.

5.2.3. Publicaciones previas al inicio del doctorado, pero relacionadas

- Rafael Leira, Pedro Gomez Nieto, Iván González, Francisco Javier Gómez Arribas. Quality of Experience Engineering for Customer Added Value Services: From Evaluation to Monitoring, chapter 6. Iste Publishing Company, 2014. [155].
  Relación con la tesis: Forma parte del Capítulo 4.

- Rafael Leira, Pedro Gomez Nieto, Iván González, Francisco Javier Gómez Arribas. Multimedia flow classification at 10 gbps using acceleration techniques on commodity hardware. 4th IEEE Technical CoSponsored International Conference on Smart Communications in Network Technologies 2013 (SaCoNeT 2013) [156].
  Ranking de Conferencia: Sin Ranking
  Relación con la tesis: Forma parte del Capítulo 4.

5.3. Trabajo Futuro

Dados los resultados obtenidos, así como la experiencia y los conocimientos adquiridos durante la realización de esta tesis se plantean las siguientes líneas de trabajo futuras:

- **Mejora de mediciones activas**: Los sistemas de medición activos presentados en la Sección 2.2 presentan unas mediciones de alta calidad, pero a costa de utilizar un ancho de banda elevado. Para poder reducir el ancho de banda, una interesante vía de investigación consiste en intentar utilizar técnicas estadísticas más avanzadas –nuevas o ya existentes– y adaptarlas a la problemática y limitaciones que presenta la medición en enlaces de 10 y 100 Gbit/s Ethernet. Siempre, intentando mantener la resolución de los nanosegundos alcanzada en [18].

- **Continuación de Wormhole**: El sistema de procesamiento en tiempo real presentando en la Subsección 3.2.4 presenta resultados a nivel de rendi-

miento por encima de cualquier otra herramienta del estado del arte, no obstante, presenta carencias que deben ser solventadas con un mayor trabajo:

- *Mantenimiento y visibilidad*: El proyecto debe ser mantenido por la comunidad, y, por tanto, debe ser empujado y publicitado para lograr dicho cometido
- *Gestionalibidad*: La gestionalibidad de Wormhole es reducida, ya que a pesar de la configurabilidad la interfaz con el usuario son ficheros con una baja documentación. Crear una mejor documentación, así como una interfaz de gestión equiparable al estado del arte, es fundamental.
- *Coordinación entre nodos*: Los sistemas de coordinación entre nodos y adaptación dinámica se encuentran en pañales comparados con algunos modelos del estado del arte. Nuevas aproximaciones que fuesen capaz de autodimensionar y escalar un flujo de acuerdo a un conjunto de parámetros es fundamental para hacer a Wormhole una herramienta competitiva.

- **Análisis de seguridad mediante Machine Learning**: El uso de algoritmos de aprendizaje automático para la detección de brechas de seguridad, así como de comportamientos anómalos, es una vía de investigación por sí sola, en la cual se pretende profundizar poco a poco aplicando un mayor número de algoritmos y nuevos parámetros de red, que nos permitan ampliar los resultados obtenidos y discutidos en el Capítulo 4.

- **Análisis sobre coprocesadores**: El procesamiento y análisis de tráfico es complejo y el uso de coprocesadores es vital para poder realizar ciertos análisis especialmente complejos, tal y como se ha explicado en la Sección 4.2. No obstante, el coste de desarrollo e implantación de este tipo de soluciones evita su puesta en marcha en determinados sitios. Reducir el coste, así como encontrar soluciones más eficientes es una linea de investigación continua, y en la cual se ha avanzado mediante la dirección de un trabajo fin de máster [157].

- **Diplodokus**: Probablemente el final de esta tesis y su camino natural ha sido la creación del proyecto *Naudit Diplodokus*. Un sistema de monitorización completo basado en DPDK que va a sustituir a la tecnología HPCAP

actualmente utilizada. El proyecto pretende utilizar la mayor parte de las experiencias aprendidas a lo largo de la tesis en su desarrollo, así como el trabajo y experiencia de otros investigadores de Naudit HPCN S.L. A fecha de finalización de esta tesis, la previsión de entrar en producción es a principios del año 2020.

Acrónimos

API Application Programming Interface. 15–18, 20, 26, 33, 54, 88

ASIC Application-Specific Integrated Circuit. 11

CAPEX Capital expenditures. III, VII, 3, 21

CPD centro de procesamiento de datos. 2

DMA Direct Memory Access. 16, 108

DPDK Data Plane Development Kit. XVII, 18–20, 27, 30, 32, 37–48, 52, 54–58, 63, 64, 68, 84, 88, 92, 135, 140

FC Fibre Channel. 50

FCoE Fibre Channel sobre Ethernet. 50, 53

FPGA Field Programmable Gate Array. 11, 12, 30, 52, 92, 106, 108–110, 112, 114, 115

GPU Graphics Processing Unit. 14, 15, 32, 106, 108, 112, 114, 115

IOT Internet Of Things. 72

IP Internet Protocol. 1, 7, 10, 39, 55, 96, 98, 100, 101, 103, 117, 119

iSCSI Internet SCSI. 50

ISP operadora de Internet. 2, 10, 21

KVM Kernel Virtual Machine. 23, 26, 31, 41, 45–48

MAC Media Access Control. 39

MIB Management Information Base. 8

NFV Network Function Virtualization. 19, 21–23, 29, 30

NVMe Non-Volatile Memory Express. 37, 41, 43, 47, 89, 135

OAA Observar, Analizar y Actuar. 21

OPEX Operational Expenditures. III, VII, 3, 21

OvS Open virtual Switch. 26, 27, 29

PF Physical Function. 24, 28, 38, 39, 41

RAID Redundant Array of Independent Disks. 32, 33, 37, 41, 43, 47, 135

RSS Receive Side Scaling. 15, 18, 33–35, 84, 90, 92

RTT Round-Trip Time. 49, 50, 66, 86, 87, 119

SAN Storage Area Network. 49–51, 53

SDN Software Defined Networking. 19, 21, 29, 88

SNMP Simple Network Management Protocol. 8

SPDK Storage Performance Development Kit. 37, 47, 135

VF Virtual Function. XV, 22–24, 28, 38–41, 44–46, 68

VM Virtual Machine. 24–26, 28, 31, 39, 40, 42–45, 85, 86

VNP Virtual Network Probe. 21, 22, 30, 40–42, 46–48, 67

Bibliografía

[1] J. Postel, "INTERNET PROTOCOL," RFC 791, Sep. 1981.

[2] C. G. C. Index, "Forecast and methodology, 2016-2021 white paper," *Retrieved 1st June*, 2016.

[3] H. Chaouchi, *The Internet of things: connecting objects to the web*. John Wiley & Sons, 2013.

[4] M. Mellia, R. L. Cigno, and F. Neri, "Measuring ip and tcp behavior on edge nodes with tstat," *Computer Networks*, vol. 47, no. 1, pp. 1–21, 2005.

[5] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and S. C. Diot, "Packet-level traffic measurements from the sprint ip backbone," *IEEE network*, vol. 17, no. 6, pp. 6–16, 2003.

[6] P. M. Santiago del Río, "Internet traffic classification for high-performance and off-the-shelf systems," Ph.D. dissertation, Universidad Autónoma de Madrid, 2013.

[7] V. M. Martínez, "Harnessing low-level tuning in modern architectures for high-performance network monitoring in physical and virtual platforms," Ph.D. dissertation, Universidad Autónoma de Madrid, 2015.

[8] X. Zhou, H. Liu, and R. Urata, "Datacenter optics: requirements, technologies, and trends," *Chinese Optics Letters*, vol. 15, no. 5, p. 120008, 2017.

[9] J. Yu and X. Zhou, "Ultra-high-capacity dwdm transmission system for 100g and beyond," *IEEE Communications Magazine*, vol. 48, no. 3, pp. S56–S64, March 2010.

[10] Intel Corporation, "Intel 64 and IA-32 architectures optimization reference manual," January 2016, http://www.intel.co.uk/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[11] V. Moreno, P. M. Santiago del Río, J. Ramos, D. Muelas, J. L. García-Dorado, F. J. Gomez-Arribas, and J. Aracil, "Multi-granular, multi-purpose and multi-Gb/s monitoring on off-the-shelf systems," *International Journal of Network Management*, vol. 24, no. 4, pp. 221–234, 2014.

[12] V. Moreno, J. Ramos, P. M. Santiago del Río, J. L. García-Dorado, F. J. Gomez-Arribas, and J. Aracil, "Commodity packet capture engines: Tutorial, cookbook and applicability," *IEEE Communications Surveys Tutorials*, vol. 17, no. 3, pp. 1364–1390, thirdquarter 2015.

[13] W. Nagele, "Large-scale PCAP Data Analysis Using Apache Hadoop," 2011. [Online]. Disponible en: https://labs.ripe.net/Members/wnagele/large-scale-pcap-data-analysis-using-apache-hadoop

[14] M. Zangrilli and B. B. Lowekamp, "Comparing passive network monitoring of grid application traffic with active probes," in *Proceedings. First Latin American Web Congress*, Nov 2003, pp. 84–91.

[15] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "A simple network management protocol (snmp)," RFC 1067, Aug. 1988.

[16] P. Chatzimisios, "Security issues and vuluerabilities of the snmp protocol," in *(ICEEE). 1st International Conference on Electrical and Electronics Engineering, 2004.* IEEE, 2004, pp. 74–77.

[17] L. Andrey, O. Festor, A. Lahmadi, A. Pras, and J. Schönwälder, "Survey of snmp performance analysis studies," *International Journal of Network Management*, vol. 19, no. 6, pp. 527–548, 2009.

[18] R. Leira, J. Aracil, J. E. López de Vergara, P. Roquero, and I. González, "High-speed optical networks latency measurements in the microsecond timescale with software-based traffic injection," *Optical Switching and Networking*, vol. 29, pp. 39 – 45, 2018.

[19] J. L. García-Dorado, P. M. Santiago del Río, J. Ramos, D. Muelas, V. Moreno, J. E. López de Vergara, and J. Aracil, "Low-cost and high-performance: VoIP

monitoring and full-data retention at multi-Gb/s rates using commodity hardware," *International Journal of Network Management*, vol. 24, no. 3, pp. 181–199, 2014.

[20] D. Salcedo, C. Guerrero, and J. Guérrero, "Overhead in available bandwidth estimation tools: Evaluation and analysis," *International Journal of Communication Networks and Information Security (IJCNIS)*, vol. 9, no. 3, 2017.

[21] D. Hernando-Loeda, J. E. López de Vergara, J. Aracil, D. Madrigal, and F. Mata, "Measuring mpeg frame loss rate to evaluate the quality of experience in iptv services," *Quality of Experience Engineering for Customer Added Value Services*, pp. 31–51, 2013.

[22] J. Ramos, P. Santiago del Río, J. Aracil, and J. E. López de Vergara, "On the effect of concurrent applications in bandwidth measurement speedometers," *Computer Networks*, vol. 55, no. 6, pp. 1435 – 1453, 2011.

[23] M.-S. Kim, Y. J. Won, and J. W. Hong, "Characteristic analysis of Internet traffic from the perspective of flows," *Computer Communications*, vol. 29, no. 10, pp. 1639–1652, 2006.

[24] B. Claise, "Cisco systems netflow services export version 9," RFC 3954, Oct. 2004.

[25] R. Hofstede, P. Celeda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras, "Flow monitoring explained: From packet capture to data analysis with NetFlow and IPFIX," *IEEE Communications Surveys Tutorials*, vol. 16, no. 4, pp. 2037–2064, Fourthquarter 2014.

[26] B. Claise, "Specification of the ip flow information export (ipfix) protocol for the exchange of ip traffic flow information," RFC 5101, Jan. 2008.

[27] B. Trammell and E. Boschi, "An introduction to IP flow information export (IPFIX)," *Communications Magazine, IEEE*, vol. 49, no. 4, pp. 89–95, April 2011.

[28] M. Kodialam and T. Lakshman, "Detecting network intrusions via sampling: a game theoretic approach," in *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, vol. 3, March 2003, pp. 1880–1889 vol.3.

[29] D. Brauckhoff, B. Tellenbach, A. Wagner, M. May, and A. Lakhina, "Impact of packet sampling on anomaly detection metrics," in *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '06. New York, NY, USA: ACM, 2006, pp. 159–164. [Online]. Disponible en: http://doi.acm.org/10.1145/1177080.1177101

[30] A. Pescapé, D. Rossi, D. Tammaro, and S. Valenti, "On the impact of sampling on traffic monitoring and analysis," in *Teletraffic Congress (ITC), 2010 22nd International*. IEEE, 2010, pp. 1–8.

[31] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita, "Information metrics for low-rate ddos attack detection: A comparative evaluation," in *2014 Seventh International Conference on Contemporary Computing (IC3)*, Aug 2014, pp. 80–84.

[32] M. Forconesi, G. Sutter, S. López-Buedo, and C. Sisterna, "Clasificación de flujos de comunicación en redes de 10 gbps con fpgas," in *Jornadas SARTECO 2012*, 2012. [Online]. Disponible en: http://www.jornadassarteco.org/js2012/papers/paper_64.pdf

[33] M. Forconesi, G. Sutter, S. Lopez-Buedo, J. E. Lopez de Vergara, and J. Aracil, "Bridging the gap between hardware and software open-source network developments," *IEEE Network*, vol. 28, no. 5, 2014.

[34] J. J. Garnica, S. Lopez-Buedo, V. Lopez, J. Aracil, and J. M. G. Hidalgo, "A FPGA-based scalable architecture for URL legal filtering in 100GbE networks." in *ReConFig*, 2012, pp. 1–6.

[35] M. Kang, D.-I. Kang, M. Yun, G.-L. Park, and J. Lee, "Design for run-time monitor on cloud computing," in *Security-Enriched Urban Computing and Smart Grid*. Springer, 2010, pp. 279–287.

[36] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: a GPU-accelerated software router," in *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4. ACM, 2010, pp. 195–206.

[37] L. Deri *et al.*, "Improving passive packet capture: Beyond device polling," in *Proceedings of SANE*, vol. 2004. Amsterdam, Netherlands, 2004, pp. 85–93.

[38] ntop, "Libzero for DNA," 2014, www.ntop.org/products/pf_ring/libzero-for-dna/, [1 August 2014].

[39] S. Woo and K. Park, "Scalable tcp session monitoring with symmetric receive-side scaling," *KAIST, Daejeon, Korea, Tech. Rep*, 2012.

[40] L. Rizzo, "Netmap: a novel framework for fast packet I/O," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 101–112.

[41] L. Rizzo, L. Deri, and A. Cardigliano, "10 Gbit/s line rate packet processing using commodity hardware: survey and new proposals," 2012, [Accedido el 1-Septiembre-2019]. [Online]. Disponible en: http://luca.ntop.org/10g.pdf

[42] V. Moreno, "Development and evaluation of a low-cost scalable architecture for network traffic capture and storage for 10Gbps networks," Master's thesis, Universidad Autónoma de Madrid, 2012.

[43] *DPDK: Programmer's Guide*, Apr. 2018, https://fast.dpdk.org/doc/pdf-guides-18.05/.

[44] R. Jain and S. Paul, "Network virtualization and software defined networking for cloud computing: a survey," *IEEE Communications Magazine*, vol. 51, no. 11, pp. 24–31, 2013.

[45] S. van der Meer, J. Keeney, and L. Fallon, "5g networks must be autonomic!" in *IEEE/IFIP Network Operations and Management Symposium (NOMS 2018)*, Apr. 2018.

[46] L. Velasco, L. Gifre, J. L. Izquierdo-Zaragoza, F. Paolucci, A. P. Vela, A. Sgambelluri, M. Ruiz, and F. Cugini, "An architecture to support autonomic slice networking," *Journal of Lightwave Technology*, vol. 36, no. 1, pp. 135–141, Jan 2018.

[47] L. Gifre, A. P. Vela, M. Ruiz, J. L. de Vergara, and L. Velasco, "Experimental assessment of node and control architectures to support the observe-analyze-act loop," in *Optical Fiber Communication Conference*. Optical Society of America, 2017, pp. Th1J–3.

[48] "Minimum requirements related to technical performance for IMT-2020 radio interface(s)," ITU-R, Tech. Rep. SG05 Contribution 40, Feb. 2017.

[49] "Ieee standard for ethernet - amendment 10: Media access control parameters, physical layers, and management parameters for 200 gb/s and 400 gb/s operation," *IEEE Std 802.3bs-2017 (Amendment to IEEE 802.3-2015 as amended by IEEE's 802.3bw-2015, 802.3by-2016, 802.3bq-2016, 802.3bp-2016, 802.3br-2016, 802.3bn-2016, 802.3bz-2016, 802.3bu-2016, 802.3bv-2017, and IEEE 802.3-2015/Cor1-2017)*, pp. 1–372, Dec 2017.

[50] G. Karagiannis, D. Hai, G. Dobrowski, Y. Hertoghs, and N. Zong, "Cloud central office reference architectural framework," Broadband forum, Tech. Rep. TR-384, Jan. 2018.

[51] S. C. Ugbolue, *Polyolefin Fibres: Structure, Properties and Industrial Applications*. Woodhead Publishing, 2017, chapter 2.4: "Monitoring-as-a-Service (MaaS)".

[52] Altnix, "Monitoring as a service (MaaS) - does it work?" White Paper, Altnix, Feb. 2014.

[53] M. Jarschel, T. Zinner, T. Höhn, and P. Tran-Gia, "On the accuracy of leveraging sdn for passive network measurements," in *2013 Australasian Telecommunication Networks and Applications Conference (ATNAC)*, Nov 2013, pp. 41–46.

[54] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[55] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2015, pp. 171–172. [Online]. Disponible en: http://domino.watson.ibm.com/library/CyberDig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf

[56] L. Rizzo, G. Lettieri, and V. Maffione, "Speeding up packet I/O in virtual machines," in *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2013.

[57] R. Russell, "Virtio: Towards a de-facto standard for virtual I/O devices," *SIGOPS Operating Systems Review*, vol. 42, no. 5, 2008.

[58] G. Motika and S. Weiss, "Virtio network paravirtualization driver: Implementation and performance of a de-facto standard," *Computer Standards & Interfaces*, vol. 34, no. 1, 2012.

[59] J. Hwang, K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms," in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[60] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vswitch," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015, pp. 117–130. [Online]. Disponible en: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff

[61] P. Emmerich, D. Raumer, S. Gallenmüller, F. Wohlfart, and G. Carle, "Throughput and latency of virtual switching with open vswitch: A quantitative analysis," *Journal of Network and Systems Management*, vol. 26, no. 2, pp. 314–338, Apr 2018. [Online]. Disponible en: https://doi.org/10.1007/s10922-017-9417-0

[62] C. Yang, J. Liu, H. Wang, and C. Hsu, "Implementation of gpu virtualization using pci pass-through mechanism," *The Journal of Supercomputing*, vol. 68, no. 1, 2014.

[63] V. Moreno, R. Leira, I. Gonzalez, and F. J. Gomez-Arribas, "Towards high-performance network processing in virtualized environments," in *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, vol. 12. IEEE, 2015, pp. 535–540.

[64] S. Yamany, "Methods, systems, and computer program products for distributed packet traffic performance analysis in a communication network," Feb. 7 2017, uS Patent 9,565,073.

[65] F. Moradi, C. Flinta, A. Johnsson, and C. Meirosu, "Conmon: an automated container based network performance monitoring system," in *Integrated Network and Service Management (IM), 2017 IFIP/IEEE Symposium on*. IEEE, 2017, pp. 54–62.

[66] H. Mahkonen, R. Manghirmalani, M. Shirazipour, M. Xia, and A. Takacs, "Elastic network monitoring with virtual probes," in *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, Nov 2015, pp. 1–3.

[67] E. J. Halpern and E. C. Pignataro, "Service function chaining (sfc) architecture," RFC 7665, Oct. 2015.

[68] G. Brown, "Virtual probes for nfvi monitoring," White Paper, Intel and Qosmos a division of ENEA, Feb. 2017.

[69] S. Shanmugalingam, A. Ksentini, and P. Bertin, "Dpdk open vswitch performance validation with mirroring feature," in *2016 23rd International Conference on Telecommunications (ICT)*, May 2016, pp. 1–6.

[70] J. F. Zazo, S. Lopez-Buedo, Y. Audzevich, and A. W. Moore, "A pcie dma engine to support the virtualization of 40 gbps fpga-accelerated network appliances," in *ReConFigurable Computing and FPGAs (ReConFig), 2015 International Conference on*. IEEE, 2015, pp. 1–6.

[71] M. Helsley, "Lxc: Linux container tools," *IBM devloperWorks Technical Library*, vol. 11, 2009.

[72] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux Symposium*, 2007.

[73] M. Raho, A. Spyridakis, M. Paolino, and D. Raho, "Kvm, xen and docker: A performance analysis for arm based nfv and cloud computing," in *2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, Nov 2015, pp. 1–8.

[74] A. Kudryavtsev, V. Koshelev, and A. Avetisyan, "Prospects for virtualization of high-performance x64 systems," *Programming and Computer Software*, vol. 39, no. 6, 2013.

[75] G. Julián-Moreno, R. Leira, J. E. López de Vergara, F. J. Gomez-Arribas, and I. Gonzalez, "On the feasibility of 40 gbps network data capture and retention with general purpose hardware," in *33rd ACM/SIGAPP Symposium On Applied Computing (SAC'2018)*.

[76] V. Moreno, P. Santiago del Rio, J. Ramos, J. Garcia-Dorado, I. Gonzalez, F. Gomez-Arribas, and J. Aracil, "Packet storage at multi-gigabit rates using off-the-shelf systems," in *Proceedings of the IEEE International Conference on High Performance and Communications (HPCC2014)*, 2014.

[77] C. Walsworth, E. Aben, k. Claffy, and D. Andersen, "The CAIDA anonymized Internet traces 2016 dataset," http://www.caida.org/data/passive/passive_2016_dataset.xml, [06 April 2016].

[78] P. Emmerich, M. Pudelko, S. Gallenmüller, and G. Carle, "Flowscope: Efficient packet capture and storage in 100 gbit/s networks," in *Proceedings of the 16th International IFIP TC6 Networking Conference, IEEE*, 2017.

[79] W. de Vries, "Scalable high-speed packet capture," in *RIPE 71*, Nov. 2015.

[80] A. Johnsson, "On the comparison of packet-pair and packet-train measurements," in *Proc. Swedish National Computer Networking Workshop*, 2003, pp. 241–250.

[81] M. Ruiz, J. Ramos, G. Sutter, J. E. López de Vergara, S. López-Buedo, and J. Aracil, "Accurate and affordable packet-train testing systems for multi-gigabit-per-second networks," *IEEE Communications Magazine*, vol. 54, no. 3, pp. 80–87, March 2016.

[82] A. Tockhorn, P. Danielis, and D. Timmermann, "A configurable FPGA-based traffic generator for high-performance tests of packet processing systems," in *6th International Conference on Internet Monitoring and Protection (ICIMP)*, 2011, pp. 14–19.

[83] Y. Wang, Y. Liu, X. Tao, and Q. He, "An FPGA-based high-speed network performance measurement for RFC 2544," *EURASIP Journal on Wireless Communications and Networking*, vol. 2015, no. 1, p. 2, 2015.

[84] J. W. Lockwood and M. Monga, "Implementing ultra low latency data center services with programmable logic," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, Aug 2015, pp. 68–77.

[85] R. Olsson, "pktgen the linux packet generator," in *Proceedings of the Linux Symposium, Ottawa, Canada*, vol. 2, 2005, pp. 11–24.

[86] D. Turull, P. Sjödin, and R. Olsson, "Pktgen: Measuring performance on high speed networks," *Computer Communications*, vol. 82, pp. 39 – 48, 2016.

[87] A. BeifuSS, T. M. Runge, D. Raumer, P. Emmerich, B. E. Wolfinger, and G. Carle, "Building a low latency linux software router," in *2016 28th International Teletraffic Congress (ITC 28)*, vol. 01, Sept 2016, pp. 35–43.

[88] S. Ma, J. Kim, and S. Moon, "Exploring low-latency interconnect for scaling out software routers," in *2016 2nd IEEE International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*, March 2016, pp. 9–15.

[89] C. Wang, O. Spatscheck, V. Gopalakrishnan, Y. Xu, and D. Applegate, "Toward high-performance and scalable network functions virtualization," *IEEE Internet Computing*, vol. 20, no. 6, pp. 10–20, Nov 2016.

[90] S. Han, K. Jang, K. Park, and S. Moon, "Building a single-box 100 gbps software router," in *2010 17th IEEE Workshop on Local Metropolitan Area Networks (LANMAN)*, May 2010, pp. 1–4.

[91] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*. ACM, 2015, pp. 275–287.

[92] J. Curtis and T. Mcgregor, "Review of bandwidth estimation techniques," in *In New Zealand Computer Science Research Students Conference*, 2001.

[93] S. Chaudhuri, U. Dayal, and V. Narasayya, "An overview of business intelligence technology," *Communications of the ACM*, vol. 54, no. 8, pp. 88–98, 2011.

[94] C. Vega, P. Roquero, R. Leira, I. Gonzalez, and J. Aracil, "Loginson: a transform and load system for very large-scale log analysis in large it infrastructures," *The Journal of Supercomputing*, vol. 73, no. 9, pp. 3879–3900, 2017.

[95] C. G. Vega Moreno, "Explorando el proceso de recolección, análisis y visualización del tráfico en las redes de computadoras," Ph.D. dissertation, Universidad Autónoma de Madrid, 2018.

[96] R. Leira, L. Gifre, I. González, J. E. L. de Vergara, and J. Aracil, "Wormhole: a novel big data platform for 100 gbit/s network monitoring and beyond," in *2019 22nd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, Feb 2019, pp. 297–301.

[97] R. Gerhards, "The syslog protocol." RFC 5424, Mar. 2009.

[98] B. Azarmi, "The big (data) problem," in *Scalable Big Data Architecture*. Springer, 2016, pp. 1–16.

[99] C. Kalantzis, "Revisiting 1 million writes per second," 2016, [Accedido el 1-Septiembre-2019]. [Online]. Disponible en: http://techblog.netflix.com/2014/07/revisiting-1-million-writes-per-second.html

[100] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, May 2010, pp. 1–10.

[101] E. Silva, "Fluentd: a high performance unified logging layer," 2015, [Accedido el 24-Septiembre-2018]. [Online]. Disponible en: https://www.linux.com/news/fluentd-high-performance-unified-logging-layer

[102] ——, "Unifying events & logs into the cloud," 2015, [Accedido el 24-Septiembre-2018]. [Online]. Disponible en: https://events.static.linuxfound.org/sites/events/files/slides/unifying_events.pdf

[103] "Fluentd architecture," [Accedido el 1-Septiembre-2019]. [Online]. Disponible en: https://www.fluentd.org/architecture

[104] ScyllaDB, "ScyllaDB AWS i3 Performance Benchmark," [Accedido el 1-Septiembre-2019]. [Online]. Disponible en: https://www.scylladb.com/product/benchmarks/aws-i3-metal-benchmark/

[105] V. Pu, P. Velan, L. Kekely, J. Koenek, and P. Minaík, "Hardware accelerated flow measurement of 100 gb ethernet," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, May 2015, pp. 1147–1148.

[106] S. Campbell and J. Lee, "Prototyping a 100g monitoring system," in *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Feb, 2012, pp. 293–297.

[107] M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. S. Baveja, "Lambda architecture for cost-effective batch and speed big data processing," in *2015 IEEE International Conference on Big Data (Big Data)*, Oct 2015, pp. 2785–2792.

[108] Y. Lee and Y. Lee, "Toward scalable internet traffic measurement and analysis with hadoop," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 1, pp. 5–13, 2013.

[109] F. Garillot, *Learning Spark Streaming*. O'Reilly Media, Inc., 2017.

[110] P. Hintjens, *ZeroMQ: messaging for many applications*. O'Reilly Media, Inc., 2013.

[111] N. Narkhede, G. Shapira, and T. Palino, *Kafka: The Definitive Guide: Real-time Data and Stream Processing at Scale*. O'Reilly Media, Inc., 2017.

[112] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 1789–1792.

[113] M. A. Lopez, A. G. P. Lobato, and O. C. M. B. Duarte, "A performance comparison of open-source stream processing platforms," in *2016 IEEE Global Communications Conference (GLOBECOM)*, Dec 2016, pp. 1–6.

[114] W. Inoubli, S. Aridhi, H. Mezni, M. Maddouri, and E. M. Nguifo, "An experimental survey on big data frameworks," *Future Generation Computer Systems*, 2018.

[115] A. Jain, *Mastering Apache Storm*. Packt Publishing Ltd, 2017.

[116] F. Hueske and V. Kalavri, *Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications*. O'Reilly Media, Inc., 2017.

[117] J. Kreps, "Benchmarking Apache Kafka: 2 million writes per second (on three cheap machines)," https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines, 2014.

[118] Y. Wang, C. Xu, X. Li, and W. Yu, "Jvm-bypass for efficient hadoop shuffling," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, May 2013, pp. 569–578.

[119] J. Cho, H. Chang, S. Mukherjee, T. V. Lakshman, and J. Van der Merwe, "Typhoon: An sdn enhanced real-time big data streaming framework," in *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '17. New York, NY, USA: ACM, 2017, pp. 310–322. [Online]. Disponible en: http://doi.acm.org/10.1145/3143361.3143398

[120] M. Ruiz, G. Sutter, S. López-Buedo, and J. E. L. de Vergara, "Fpga-based encrypted network traffic identification at 100 gbit/s," in *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Nov 2016, pp. 1–6.

[121] "Apache Hadoop." [Online]. Disponible en: http://hadoop.apache.org

[122] J. Dean and s. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, January 2008.

[123] S. Ghemawat, H. Gobioff, and S. Leung, "The Google File System," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 29–43, December 2003.

[124] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop YARN: yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[125] "Institutions using Hadoop." [Online]. Disponible en: https://wiki.apache.org/hadoop/PoweredBy

[126] "Products that include Hadoop." [Online]. Disponible en: http://wiki.apache.org/hadoop/Distributions%20and%20Commercial%20Support

[127] Y. Lee, W. Kang, and Y. Lee, "A Hadoop-based Packet Trace Processing Tool," in *Proceedings of the Third International Conference on Traffic Monitoring and Analysis*, ser. TMA'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 51–63. [Online]. Disponible en: http://dl.acm.org/citation.cfm?id=1986282.1986289

[128] "Message Headers." [Online]. Disponible en: http://www.iana.org/assignments/message-headers/message-headers.xml

[129] C. G. Vega Moreno, "Disección de tráfico web a alta velocidad," Master's thesis, Universidad Autónoma de Madrid, Sep. 2014.

[130] M. Roesch, "Snort - lightweight intrusion detection for networks," ser. LISA '99. Berkeley, CA, USA: USENIX Association, 1999, pp. 229–238.

[131] A. W. Moore and D. Zuev, "Internet traffic classification using bayesian analysis techniques," *SIGMETRICS Perform. Eval. Rev.*, vol. 33, no. 1, pp. 50–60, Jun. 2005.

[132] N. Cascarano, A. Este, F. Gringoli, F. Risso, and L. Salgarelli, "An experimental evaluation of the computational cost of a dpi traffic classifier," in *Proceedings of the 28th IEEE conference on Global telecommunications*, ser. GLOBECOM'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 1132–1139.

[133] M. Najam, U. Younis, and R. Rasool, "Multi-byte Pattern Matching Using Stride-K DFA for High Speed Deep Packet Inspection," in *2014 IEEE 17th International Conference on Computational Science and Engineering (CSE)*, Dec. 2014, pp. 547–553.

[134] W. Melo, S. Fernandes, R. Antonello, D. Sadok, J. Kelner, and G. Szabo, "A look under the hood: Revealing performance issues in the DPI engine," in *2013 IEEE International Conference on Communications (ICC)*, Jun. 2013, pp. 2974–2978.

[135] K. Neshatpour, M. Malik, and H. Homayoun, "Accelerating machine learning kernel in hadoop using fpgas," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2015, pp. 1151–1154.

[136] J. Zhu, J. Li, E. Hardesty, H. Jiang, and K.-C. Li, "Gpu-in-hadoop: Enabling mapreduce across distributed heterogeneous platforms," in *2014 IEEE/ACIS 13th International Conference on Computer and Information Science (ICIS)*. IEEE, 2014, pp. 321–326.

[137] D. Yang, X. Zhong, D. Yan, F. Dai, X. Yin, C. Lian, Z. Zhu, W. Jiang, and G. Wu, "Nativetask: a hadoop compatible framework for high performance," in *2013 IEEE International Conference on Big Data*. IEEE, 2013, pp. 94–101.

[138] R. Leira Osuna, P. Gómez Nieto, I. González Vidal, and J. E. López de Vergara, "High speed multimedia flow classification," *Quality of Experience Engineering for Customer Added Value Services*, pp. 93–118, Jul. 2014.

[139] G. Antichi, M. Shahbaz, Y. Geng, N. Zilberman, A. Covington, M. Bruyere, N. McKeown, N. Feamster, B. Felderman, M. Blott, A. Moore, and P. Owezarski, "OSNT: open source network tester," *IEEE Network*, vol. 28, no. 5, pp. 6–12, Sep. 2014.

[140] N. Cascarano, L. Ciminiera, and F. Risso, "Optimizing Deep Packet Inspection for High-Speed Traffic Analysis," *J. Netw. Syst. Manage.*, vol. 19, no. 1, pp. 7–31, Mar. 2011.

[141] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[142] Cisco, "Tetration," [Accedido el 15-Octubre-2019]. [Online]. Disponible en: https://www.cisco.com/c/en/us/support/docs/data-center-analytics/tetration-analytics/214212-how-to-monitor-your-tetration-cluster.html#anc6

[143] "Grafana." [Online]. Disponible en: https://grafana.com/

[144] "Kibana." [Online]. Disponible en: https://www.elastic.co/es/products/kibana

[145] R. Leira, I. González, L. Gifre, and J. E. L. de Vergara, "Network analysis on lambda architecture," Nov. 2017.

[146] R. Leira, G. Julian-Moreno, I. González, F. J. Gomez-Arribas, and J. E. L. de Vergara, "On the performance assessment of 40 gbit/s off-the-shelf net-

work cards for virtual network probes in 5g networks," *Computer Networks*, vol. 152, pp. 133–143, 2019.

[147] R. Leira, P. Roquero, C. Vega, I. Gonzalez, and J. Aracil, "Hpsengine: Motor de alto rendimiento y bajalatencia para el procesamiento distribuido entiempo real," in *XXVI Jornadas de Paralelismo*, Salamanca, Spain, Sep 2016.

[148] J. R. de Santiago and J. A. Rico, "Proactive measurement techniques for network monitoring in heterogeneous environments," Ph.D. dissertation, Universidad Autónoma de Madrid, 2013.

[149] "2018 Fabio Neri Best Paper Award," 2018, [Accedido el 29-August-2019]. [Online]. Disponible en: https://www.journals.elsevier.com/optical-switching-and-networking/awards/congratulations-to-the-winners-of-the-2018

[150] "Gii-grin-scie (ggs) conference rating." [Online]. Disponible en: http://gii-grin-scie-rating.scie.es/

[151] A. N. de Evaluación de la Calidad y Acreditación (ANECA), "Criterios de evaluación," pp. 9,17,41,47, 2017, . [Online]. Disponible en: http://www.aneca.es/content/download/13783/171471/file/CRITERIOS_INGENIERIA_ARQUIT.pdf

[152] J. Fernando Zazo, R. Martín, R. Leira, I. González, G. Sutter, and F. J. Gomez Arribas, "Clasificación de tráfico de red mediante aceleradores hardware," in *XV Jornadas de Computación Reconfigurable y Aplicaciones 2015*, Córdoba, Spain, Sep 2015.

[153] R. García-Valcárcel, R. Leira, I. Gonzalez, and J. E. López de Vergara, "Monitorización y análisis de tráfico de red con apache hadoop," in *Jornadas de Ingeniería Telemática (JITEL)*, Palma de Mayorca, Spain, Oct 2015.

[154] R. García-Valcárcel, R. Leira, I. Gonzalez, and F. J. Gomez Arribas, "Evaluando apache hadoop para análisis de tráfico de red," in *XXV Jornadas de Paralelismo*, Córdoba, Spain, Sep 2015.

[155] R. Leira Osuna, P. Gomez Nieto, I. Gonzalez, and J. Lopez de Vergara, *Quality of Experience Engineering for Customer Added Value Services: From Evaluation to Monitoring*. Iste Publishing Company, 2014, ch. 6.

[156] R. Leira, P. Gómez, I. González, and J. E. López de Vergara, "Multimedia flow classification at 10 Gbps using acceleration techniques on commodity hardware," in *2013 International Conference on Smart Communications in Network Technologies (SaCoNeT)*, vol. 03, June 2013, pp. 1–5.

[157] A. Lopez Ibañez, "Diseño de una tarjeta aceleradora basada en fpga para el procesamiento de tráfico en redes 10 gbps ethernet," Master's thesis, 2018.